C Compiler

User's Guide and Reference for the 68000 Family

100131-016

TRADEMARKS

ARTX/ADA Realtime Executive®, Microtec®, the Microtec logo, Nanokernel®, RTscope®, RTsource®, Spectra®, VRTX®, VRTX32®, VRTX*velocity*®, XRAY®, Xtrace®, and Xtrace® Protocol are registered trademarks of Microtec.
BSPBuilder™, FastStart™, IFX™, KernelBuilder™, KernelIntegrator™, *logio*™, SNX™, Source Explorer™, the Spectra logo, Target Manager™, TNX™, ToolBuilder™, Virtual Target™, VRTX*mc*™, VRTX/OS™, VRTX*sa*™, Xconfig™, Xpert™, Xpert Profiler™, XRAY In-Circuit Debugger™, XRAY In-Circuit Debugger Monitor™, and XSH™ are trademarks of Microtec.

Other product names mentioned are trademarks or registered trademarks of their respective companies.

RESTRICTED RIGHTS LEGEND

U.S. Government Restricted Rights. This product and related documentation have been developed entirely at private expense and are commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the product pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Microtec
880 Ridder Park Dr.
San Jose, CA 95131

Copyright © 1987-1997 Microtec. All rights reserved. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Microtec.

00

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	08/88	T.P.
-002	Corrections.	11/88	T.P.
-003	Revise to meet ANSI C standards.	04/89	T.P
-004	(Non-ANSI) Produced at 80% for new packaging.	12/89	T.P.
-005	ANSI MCC68K.	12/89	T.P.
-006	Added instructions for the VAX/VMS, Apollo DOMAIN/IX, and Motorola Delta Systems UNIX/System V systems to the documentation set.	02/90	T.P.
-007	Added support for VAX/ULTRIX and HP-UX.	04/90	J.A.
-008	Manual revision to support software Version 4.1.	06/90	T.P.
-009	Manual production upgraded from Framemaker version 1.3 to 2.0.	09/90	T.P.
-010	Manual revised to support version 4.2 software.	09/91	L.W.
-011	Updated for 4.3 software.	11/92	L.L.
-012	Updated for 4.4 software.	12/93	D.C.
-013	Updated for 4.5 software.	12/94	J.Y.
-015	DOS Installation section removed and User's Guide and Reference manuals combined.	12/96	M.G.
-016	Updated to fix documentation bugs for latest release.	8/97	D.G.

Revision History

Preface

About This Manual

This manual serves as both a user's guide and reference for the Microtec C compiler for the 68000 family of microprocessors. The *User's Guide* section describes how to use the C compiler and provides host computer, target-specific, and operating system-specific descriptions of C compiler syntax. The *Reference* section describes features and concepts that are unique to the C language and to the Microtec C compiler.

The *User's Guide* section of this manual is organized as follows:

- Chapter 1, *Introduction*, describes the components of the MCC68K Compiler.
- Chapter 2, *UNIX/DOS User's Guide*, describes the use of the MCC68K Compiler on UNIX and UNIX-like operating systems and the DOS operating system.

The *Reference* section of this manual is organized as follows:

- Chapter 3, *The Preprocessor*, describes the compiler's predefined symbols and preprocessor directives.
- Chapter 4, *C Library Customizer*, describes the Microtec Library Customizer.
- Chapter 5, *Library Extensions*, describes the run-time library functions available with the MCC68K Compiler.
- Chapter 6, *Internal Data Representation*, describes the data type operations used by the MCC68K Compiler.
- Chapter 7, *C Calling Conventions*, describes the assembly language calling conventions for the MCC68K Compiler. This chapter helps you write assembly language routines that interface with C language functions.
- Chapter 8, *Run-Time Organization*, describes the memory organization of C programs for 68000 family microprocessor-based systems.
- Chapter 9, *Embedded Environments*, provides information to assist you when using embedded systems. It includes intermixing C code with assembler statements, using the linker, and generating reentrant code. It also

provides the necessary information for interfacing with assembly language routines.

- Chapter 10, *Optimizations*, describes how you can make your programs run faster.
- Chapter 11, *Compiler Output Files*, describes the output files produced by the MCC68K Compiler.

The *Appendix* section of this manual is organized as follows:

- Appendix A, *Error Statements,* describes error and diagnostic messages produced by the MCC68K Compiler.

Microprocessor References

The 68000 family of microprocessors includes the 68000, 68008, 68010, 68012, 68020, 68030, 68302, 68040, 68060, 68881, 68882, 68851, 68HC000, 68HC001, 68EC000, 68EC020, 68EC030, 68EC040, 68EC060, and the CPU32/CPU32+ families: 68330, 68331, 68332, 68333, 68340, 68349, and 68360. This document refers to these microprocessors as 68000 family microprocessors.

Related Publications

This documentation is written for the experienced program developer and assumes the developer has a working knowledge of the Motorola 68000 family of microprocessors. Although it provides several useful and informative program examples, this documentation does not describe the microprocessor itself. For such information, refer to the following Motorola publications:

- *Programmer's Reference Manual*, M68000PM/UD.
- *M68000 Family Resident Structured Assembler Reference Manual,* M68KMASM/D8.
- *68000 16/32-Bit Microprocessor User's Manual*, M68000UM/AD4.
- *68020 32-Bit Microprocessor User's Manual*, MC68020UM/AD.
- *68030 32-Bit Microprocessor User's Manual*, MC68030UM/AD.
- *MC68040 32-Bit Microprocessor User's Manual,* MC68040UM/AD.
- *MC68060 32-Bit Microprocessor User's Manual,* MC68060UM/AD.
- *68851 Paged Memory Management Unit User's Manual*, MC68851UM/AD.

- *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD.
- *CPU32 Central Processor Unit Reference Manual*, CPU32RM/AD.

For further information about the Microtec extended IEEE-695 format, refer to:

- *IEEE-695 Document*, Microtec, December 1992.

This guide assumes that you are familiar with programming and with the C programming language. For general information about C, refer to the following publications:

- Kernighan, Brian W., and Dennis M. Ritchie, *The C Programming Language,* 2nd ed., Prentice-Hall, Inc., 1988.

  This book contains an introduction to C with examples.

- Harbison, Samuel P., and Guy L. Steele, Jr., *C — A Reference Manual*, 3rd ed., Prentice-Hall Inc., 1991.

  This book contains a complete discussion of both ANSI C and traditional "K & R" C. It includes all library functions.

- Schildt, Herbert (annotator), *The Annotated ANSI C Standard — American National Standard for Programming Languages — C*, Osborne-McGraw-Hill, Inc., 1990.

  This book contains complete annotaions for the ANSI C programming standard.

For information about other Microtec 68000 family toolkit components, refer to the following Microtec publications:

- *Assembler/Linker/Librarian User's Guide and Reference for the 68000 Family*.

  This document describes how to use the ASM68K Assembler, the LNK68K Linker, and the LIB68K Librarian.

- *C++ Compiler User's Guide and Reference for the 68000 Family*.

  This document describes how to use the CCC68K C++ Compiler.

- *XRAY In-Circuit Debugger Monitor Documentation Set.*

  This documentation set describes how to use the XDM68K In-Circuit Debugger Monitor with XRAY Debugger.

Notational Conventions

This manual uses the notational conventions shown in Table P-1 (unless otherwise noted).

Table P-1. Notational Conventions

Symbol	Name	Usage
{ }	Curly Braces	Enclose a list from which you must choose an item.
[ ]	Square Brackets	Enclose optional items.
. . .	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
|	Vertical Bar	Separates alternative items in a list.
	Punctuation	Punctuation such as commas (,) and colons (:) must be entered as shown.
	`Typewriter Font`	Represents code or user input in interactive examples.
	Italics	Represents a descriptive item that should be replaced with an actual item.
	Bold	Represents elements that need to stand out from the main body of text.

Questions and Suggestions

Microtec is committed to providing its customers with quality software development and RTOS tools and support services. Our commitment continues beyond your purchase of the product throughout your development life cycle.

If you have questions or suggestions regarding this product, please contact your Microtec support representative. Contact numbers are listed on the back cover of this document.

Contents

Preface

About This Manual v
Microprocessor References vi
Related Publications vi
Notational Conventions viii
Questions and Suggestions viii

1 Introduction

Components of the Microtec MCC68K Package 1-1
MCC68K Compiler 1-1
CCC68K C++ Compiler 1-2
ASM68K Assembler 1-2
LNK68K Linker 1-2
LIB68K Object Module Librarian 1-3
XRAY68K Debugger (Optional) 1-3
XRAY Pro Debug Suite 1-4
Data Flow 1-4

2 UNIX/DOS User's Guide

Invoking the Compilation Driver 2-1
Command Line Syntax 2-1
File Name Extensions 2-2
Input and Output File Locations 2-3
Environment Variables 2-3
Setting Environment Variables for UNIX 2-5
Setting Environment Variables for DOS 2-6
Command Line Options 2-6
Command Line Option Summary 2-6
Set ANSI-Compliant Mode 2-16
Choose Address Mode for Sections 2-16
Save Comments 2-17
Produce Object File Only (Driver Option) 2-18
Define a Preprocessor Macro Name 2-18
Read Options From File (Driver Option) 2-19
Display Preprocessor Output (Driver Option) 2-19
Pass Command File to Linker (Driver Option) 2-19
Redirect Diagnostic Messages 2-19
Specify Format of Output Files 2-20

Generate Floating-Point Processor Instructions 2-21
Produce Debugging Information 2-21
Save Assembly File (Driver Option) 2-23
Support HP 64000 2-23
Add Search Path for Nonstandard Include Files 2-23
Change Search Path Order for Include Files 2-24
Add Search Path for Standard Include Files 2-25
Produce Minor Code Generation Variations 2-25
Generate Listing File 2-30
Generate Position-Independent Code and Data 2-30
Name the Sections 2-31
Optimize Code 2-33
Name the Output File (Driver Option) 2-38
Send Preprocessor Output to File (Driver Option) 2-38
Produce Code for Specified Processor 2-38
Suppress Diagnostic Messages 2-41
Pass Default Library to Linker (Driver Option) 2-42
Produce Assembler Source File (Driver Option) 2-42
Undefine a Preprocessor Macro Name 2-42
Modify Naming Conventions 2-42
Control Verbose Output Information (Driver Option) 2-44
Perform Extra Checking 2-44
Pass Options to Tools (Driver Option) 2-44
Initialize Uninitialized Global Data 2-45
Enable Microtec Extensions 2-46
Check Program Syntax (Driver Option) 2-47
Modify Alignment 2-47
Produce Minor Code Generation Variations (-Zx options) 2-48
Accept C++ Comments 2-48
Produce Function Range Information (Exception Handling) 2-48
Command Line Examples 2-48
Using Options Wisely 2-50
Option Combinations 2-50
ANSI Extensions 2-51
Using _ _STDC_ _ 2-51
Function Prototyping 2-51
ANSI and Floating-Point Variables 2-52
Microtec Extensions 2-53
typeof() Operator 2-54
Exception Handling 2-54
Customizing the Compilation Driver 2-55
Pragmas and Options 2-56
Locating Header Files 2-57

Finding Source Files 2-58
Determining Option Defaults 2-59
Producing Listing Files 2-59
Libraries 2-60
Distribution Files 2-60
The C Library 2-63
UNIX-Style System Functions 2-63
Start-Up Routine 2-64
Generating an Executable Program 2-65
Invoking the Linker 2-66

3 The Preprocessor

Overview 3-1
Predefined Symbols 3-1
Basic Preprocessor Symbols 3-1
Preprocessor Symbols for Targets 3-5
Preprocessor Symbols for Hosts 3-6
Compiler Options for Predefined Symbols 3-7
Preprocessor Macros 3-9
Preprocessor Directives 3-11
#informing — Issues an Informational Message 3-12
#pragma asm — Begins Embedded Assembly Instructions 3-13
#pragma eject — Controls Page Listing 3-14
#pragma endasm — Ends Embedded Assembly Instructions 3-15
#pragma error — Issues an Error Message and Halts Compilation 3-16
#pragma info — Issues an Informational Message 3-17
#pragma list — Controls Source Line Listings 3-18
#pragma macro — Lists Predefined Processor Symbols 3-19
#pragma option — Specifies Command Line Options 3-20
#pragma warn — Issues a Warning Message 3-21
#warning — Issues a Warning Message 3-22

4 C Library Customizer

Introduction 4-1
When to Create a Custom Library 4-1
Directories in the C Library Customizer 4-2
DOS System Requirements 4-2
Using the C Library Customizer 4-3
Step 1: Creating a Custom Configuration File 4-3
Step 2: Building a Custom Library 4-7
Step 3: Testing a Custom Library 4-8
Assessing Test Results 4-9
Debugging Custom Libraries 4-10
Using a Custom Libraries 4-11

5 Library Extensions

Overview 5-1
C Function Groups and Include Files 5-1
mriext.h 5-1
Non-Reentrant Functions and Extensions 5-3
System Functions 5-6
Library Function Definitions 5-7
close — Closes a Specified File 5-8
creat — Creates a Specified File 5-9
eprintf — Provides Formatted Print to Standard Error 5-10
_exit — Terminates a Program 5-11
fileno — Gets File Descriptor 5-12
ftoa — Converts Floating-Point Number to ASCII String 5-13
getl — Reads a Long Integer From a Stream 5-14
getw — Reads a Short Integer From a Stream 5-15
isascii — Tests for an ASCII Character 5-16
itoa — Converts Integer to ASCII String 5-17
itostr — Converts Unsigned Integer to ASCII String 5-18
lseek — Sets the Current Location in a File 5-19
ltoa — Converts Long Integer to ASCII String 5-20
ltostr — Converts Unsigned Long Integer to ASCII String 5-21
max — Returns the Greater of Two Values 5-22
memccpy — Copies Characters in Memory 5-23
memclr — Clears Memory Bytes 5-24
min — Returns the Lesser of Two Values 5-25
open — Opens a Specified File 5-26
putl — Writes a Long Integer to a Stream 5-28
putw — Writes a Short Integer to a Stream 5-29
read — Reads Bytes From a Specified File 5-30
sbrk — Allocates Memory Space 5-31
swab — Swaps Odd and Even Bytes in Memory 5-32
toascii — Converts a Byte to ASCII Format 5-33
_tolower — Converts Characters to Lowercase Without Argument Checking 5-34
_toupper — Converts Characters to Uppercase Without Argument Checking 5-35
unlink — Unlinks a File Name 5-36
write — Writes Bytes to a Specified File 5-37
zalloc — Allocates Data Space Dynamically 5-38

6 Internal Data Representation

Overview 6-1
Storage Layout 6-1
Target Parameters 6-3
Data Type Summary 6-3

Pointers 6-5
Type Conversion 6-5
Mixed Operands 6-6
Type Casting 6-7
Function Operations 6-7
Passing Prototyped Parameters Smaller Than int 6-7
Function Declaration With interrupt Keyword 6-8
Structure Operations 6-8
Structure Alignment 6-8
Determining Structure Size 6-9
Bit Fields 6-11
Alignment and Packing 6-13
Packing Bit Fields 6-13
Alignment of Bit Fields 6-14
Word Unit Straddling 6-14
Bit Field Alignment 6-15
Aggregates 6-16
Size of Aggregates 6-16
Packed Structures 6-17
Structure Padding 6-18
Structure Alignment When Default Alignments Differ 6-22
Packed Enumeration Types 6-23
Tips About Packing 6-24
Allocation of Variables 6-26
Register 6-26
Local Variables 6-27
Static Variables 6-27
Memory Allocation 6-27

7 C Calling Conventions

Overview 7-1
Parameter Passing 7-1
Setting Parameters 7-1
Setting Short Integers 7-2
Implicit Parameter for Structure/Union Return Value 7-4
Alignment of Structure/Union in Parameter Area 7-4
Function Return Values 7-4
Integer Return Values 7-5
Floating-Point Return Values (with FPU) 7-5
Floating-Point Return Values (without FPU) 7-5
Structure/Union Return Value 7-5
Popping Parameters 7-5
Implicit First Parameter for Structure/Union Return Value 7-6
Register and Stack Usage With Functions 7-6

Stack Frames 7-6
The Prologue 7-7
Local Variables in the Prologue 7-8
The Epilogue 7-8
Assembly Language Interfacing 7-9
Assembly Language Routine Example 7-10
Variable References from Assembly Routines 7-12
Interrupt Handlers 7-12

8 Run-Time Organization

Overview 8-1
Code Organization 8-1
Changing Default Addressing 8-2
Compiler-Generated Sections 8-3
code Section (Type CODE) 8-4
strings Section (Type CODE) 8-4
literals Section (Type CODE) 8-5
const Section (Type CODE) 8-5
ioports Section (Type DATA) 8-5
vars Section (Type DATA) 8-5
zerovars Section (Type DATA) 8-5
tags Section (Type DATA) 8-6

9 Embedded Environments

Embedded Applications 9-1
Considerations for Embedded Systems 9-1
In-Line Assembly 9-2
Examples 9-4
Features of Assembler In-Lining 9-5
Assigning asm to a Variable 9-5
Returning a Typed Value 9-5
Using #define for Readability 9-6
Variable Names Inside asm 9-6
#pragma asm or asm 9-7
Considerations for Assembler In-Lining 9-9
Addressing 9-9
Direct Memory Addressing 9-10
Calling a Function at an Absolute Address 9-10
Reentrant Code 9-11
A Simple Solution 9-12
Multitasking/Multi-Threaded Environments 9-13
First Approach 9-14
Second Approach 9-18
Position-Independent Code and Data 9-21

Position-Independent Versus Position-Dependent 9-21
Compiler Considerations 9-22
Programmer Considerations 9-23
Assembler and Linker Considerations 9-26
Absolute Versus Register-Relative Addressing 9-27
Absolute Addressing 9-27
PC-Relative Addressing 9-29
User-Modified Routines 9-31
User-Modified Routines for Embedded Systems 9-31
In Character Routine 9-31
Out Character Routine 9-31
Start Routine 9-31
Exit Routine 9-32
Initialization Routine 9-32
Heap Management Routine 9-33
System Functions 9-33
Removing Unneeded I/O Support 9-34
Removing Unneeded Floating-Point Support 9-35
Reserving a Register 9-35
Shared Program 9-36
Shared Data 9-36
System Data 9-37
Special Purposes 9-38
Data Initialization 9-39
Saving Initialized Variables in ROM 9-40
Using the Linker 9-40
Default Sections 9-40
Libraries 9-40
Messenger Symbols 9-41
INITDATA Command 9-41
Linker Command Example 9-42
Linker Command Example (ROM-Based System) 9-44
Interrupt Handlers 9-46

10 Optimizations

Overview 10-1
General Optimizations 10-1
Algebraic Simplification 10-1
Redundant Code Elimination 10-2
Strength Reduction 10-2
Global Optimizations 10-2
Dead Code Elimination 10-2
Factorization 10-4
Global Constant Propagation 10-4

Global Copy Propagation 10-5
Global Value Propagation 10-5
Register Allocation 10-6
Unused Definition Elimination 10-6
Loop Optimizations 10-6
Array Operator Synthesis 10-7
Loop Invariant Code Optimization 10-7
Loop Rotation 10-8
Strength Reduction and Index Simplification 10-9
Local Optimizations 10-10
Common Subexpression Elimination 10-10
Constant Folding 10-10
Generating Code for switch Statements 10-10
Redundant Load and Store Elimination 10-11
Jump Optimizations 10-11
Branch Tail Merging 10-12
Code Hoisting 10-12
Cross-Jump Optimization 10-13
Multiple Jump Optimization 10-14
Redundant Jump Elimination 10-14
Short/Long Displacement Optimization 10-15
Function In-Lining 10-15
inline Keyword 10-16
Instruction Scheduling 10-17
Machine-Dependent Optimizations 10-17
Generating Code for Function Prologue and Epilogue 10-17
In-Line Library Function Expansion 10-17
Grouping Stack Adjust Instructions 10-18
Indexing Arrays 10-18
Char Operations Where Possible 10-18

11 Compiler Output Files

Assembler Source File 11-1
Advantages of Producing an Assembly File 11-1
Variable Names 11-1
External and Public Variable Names 11-1
Static Variable Names 11-2
Line Numbers 11-2
Code and Data Sections 11-3
Compiler Output Listing 11-3

Appendix A: Error Statements

Overview A-1
Message Severity Levels A-2

Error Position Marker A-3
Suppressing Error Messages A-4
Reporting Problems A-5
Preparing a Test Case A-5
Calling Technical Support A-6
Compiler Messages A-6

Index Index-1

Figures

Figure 1-1. Cross Environment Data Flow Through the 68000 Family Toolkit 1-5
Figure 6-1. Memory Layout 6-2
Figure 6-2. Loading Depending on Processor Type 6-2
Figure 6-3. Byte Ordering in Words 6-2
Figure 6-4. Sample Bit Field Allocation (Over 8 Bytes) 6-12
Figure 6-5. Sample Bit Field Allocation (Over 2 Bytes) 6-13
Figure 6-6. Straddling Bit Fields 6-14
Figure 6-7. Packing Bit Fields (Big Endian) 6-15
Figure 6-8. Signed Bit Fields (Big Endian) 6-16
Figure 6-9. Packing of Aggregates (Big Endian) 6-17
Figure 6-10. Unpacked and Packed Structures (Big Endian) 6-18
Figure 6-11. Unpacked Structure: struct s 6-21
Figure 6-12. Structure Alignment Example 6-23
Figure 6-13. Structure three_tight_bytes 6-24
Figure 6-14. Sample Data Structure 6-26
Figure 7-1. Parameter Area of the Stack 7-2
Figure 7-2. Parameter Area and Short Integers 7-2
Figure 7-3. Parameter and Return Value Areas 7-4
Figure 7-4. Parameter Area and Structure/Union 7-4
Figure 9-1. Memory Configuration for Multi-Threaded Environments (Approach 1) 9-17
Figure 9-2. Memory Configuration for Multi-Threaded Environments (Approach 2) 9-20
Figure 9-3. Position-Independent Code (Example 1) 9-24
Figure 9-4. Position-Independent Code (Example 2) 9-26
Figure 9-5. Absolute Addressing for Code and Data 9-28
Figure 9-6. Absolute Addressing Example 9-28
Figure 9-7. PC-Relative Addressing 9-29
Figure 9-8. PC-Relative Addressing Example 9-30
Figure 9-9. Shared Programs 9-36

Figure 9-10. Shared Data 9-37
Figure 9-11. System Data 9-38
Figure 9-12. Reserved Register as a Pointer to Data 9-39
Figure 9-13. Reserved Register as a Storage Area 9-39
Figure 9-14. Memory Configuration 9-43
Figure 9-15. Memory Configuration (ROM Based) 9-45

Tables

Table P-1. Notational Conventions viii
Table 2-1. Default Input File Name Extensions 2-2
Table 2-2. Default Output File Name Extensions 2-3
Table 2-3. Default Environment Variables and Defaults 2-4
Table 2-4. UNIX/DOS Command Line Option Summary 2-7
Table 2-5. UNIX/DOS Allocation of Code and Data 2-16
Table 2-6. Interaction of -Og and -OR Options 2-37
Table 2-7. Processor Identification 2-39
Table 2-8. Common Option Combinations 2-50
Table 2-9. Keywords (Microtec Extensions) 2-54
Table 2-10. UNIX/DOS Library Use Conditions 2-61
Table 2-11. Library Use Conditions 2-63
Table 2-12. UNIX-Style System Functions Used by the MCC68K Library 2-64
Table 3-1. Basic Preprocessor Symbols 3-2
Table 3-2. Preprocessor Symbols for Targets 3-6
Table 3-3. Preprocessor Symbols for Hosts 3-6
Table 3-4. Preprocessor Symbols and Corresponding Compiler Options 3-7
Table 3-5. Predefined Preprocessor Macros 3-9
Table 3-6. Microtec Preprocessor Directives 3-11
Table 4-1. C Library Customizer Preprocessor Symbols 4-4
Table 5-1. mriext.h Functions 5-1
Table 5-2. mriext.h Macros 5-2
Table 5-3. Non-Reentrant Functions and Extensions 5-3
Table 5-4. System Functions 5-6
Table 5-5. Mode Values for the open Function 5-26
Table 6-1. Parameter Limits 6-3
Table 6-2. Scalar Data Types 6-4
Table 6-3. Complex (Aggregate) Types 6-5
Table 6-4. Mixed Operand Conversion 6-6
Table 6-5. Integral Promotion 6-6

Table 6-6. Natural Boundary Alignment for the 68000 Family 6-19
Table 6-7. Effect of Compiler Options on Structure Layout 6-22
Table 7-1. Instructions for Removing Bytes from Stack 7-7
Table 8-1. Sections Generated by the Compiler 8-1
Table 8-2. Example Modules 8-3
Table 8-3. Sections Contained in module1.o and module2.o 8-3
Table 8-4. Sections Contained in the Executable File 8-4
Table 9-1. Position-Independent Code Options 9-21
Table 9-2. Absolute Versus Register-Relative Addressing 9-27
Table 9-3. Commands for Absolute Addressing Example 9-29
Table 9-4. Commands for PC-Relative Addressing Example 9-30
Table A-1. Message Severity Levels A-2
Table A-2. Using Compiler Options to Suppress Diagnostic Messages A-4

Figure 1-1. Cross Environment Data Flow Through the 68000 Family Toolkit 1-5
Figure 6-1. Memory Layout 6-2
Figure 6-2. Loading Depending on Processor Type 6-2
Figure 6-3. Byte Ordering in Words 6-2
Figure 6-4. Sample Bit Field Allocation (Over 8 Bytes) 6-12
Figure 6-5. Sample Bit Field Allocation (Over 2 Bytes) 6-13
Figure 6-6. Straddling Bit Fields 6-14
Figure 6-7. Packing Bit Fields (Big Endian) 6-15
Figure 6-8. Signed Bit Fields (Big Endian) 6-16
Figure 6-9. Packing of Aggregates (Big Endian) 6-17
Figure 6-10. Unpacked and Packed Structures (Big Endian) 6-18
Figure 6-11. Unpacked Structure: struct s 6-21
Figure 6-12. Structure Alignment Example 6-23
Figure 6-13. Structure three_tight_bytes 6-24
Figure 6-14. Sample Data Structure 6-26
Figure 7-1. Parameter Area of the Stack 7-2
Figure 7-2. Parameter Area and Short Integers 7-2
Figure 7-3. Parameter and Return Value Areas 7-4
Figure 7-4. Parameter Area and Structure/Union 7-4
Figure 9-1. Memory Configuration for Multi-Threaded Environments (Approach 1) 9-17
Figure 9-2. Memory Configuration for Multi-Threaded Environments (Approach 2) 9-20
Figure 9-3. Position-Independent Code (Example 1) 9-24
Figure 9-4. Position-Independent Code (Example 2) 9-26
Figure 9-5. Absolute Addressing for Code and Data 9-28
Figure 9-6. Absolute Addressing Example 9-28
Figure 9-7. PC-Relative Addressing 9-29
Figure 9-8. PC-Relative Addressing Example 9-30
Figure 9-9. Shared Programs 9-36
Figure 9-10. Shared Data 9-37
Figure 9-11. System Data 9-38
Figure 9-12. Reserved Register as a Pointer to Data 9-39
Figure 9-13. Reserved Register as a Storage Area 9-39
Figure 9-14. Memory Configuration 9-43
Figure 9-15. Memory Configuration (ROM Based) 9-45

Table P-1. Notational Conventions viii
Table 2-1. Default Input File Name Extensions 2-2
Table 2-2. Default Output File Name Extensions 2-3
Table 2-3. Default Environment Variables and Defaults 2-4
Table 2-4. UNIX/DOS Command Line Option Summary 2-7
Table 2-5. UNIX/DOS Allocation of Code and Data 2-16
Table 2-6. Interaction of -Og and -OR Options 2-37
Table 2-7. Processor Identification 2-39
Table 2-8. Common Option Combinations 2-50
Table 2-9. Keywords (Microtec Extensions) 2-54
Table 2-10. UNIX/DOS Library Use Conditions 2-61
Table 2-11. Library Use Conditions 2-63
Table 2-12. UNIX-Style System Functions Used by the MCC68K Library 2-64
Table 3-1. Basic Preprocessor Symbols 3-2
Table 3-2. Preprocessor Symbols for Targets 3-6
Table 3-3. Preprocessor Symbols for Hosts 3-6
Table 3-4. Preprocessor Symbols and Corresponding Compiler Options 3-7
Table 3-5. Predefined Preprocessor Macros 3-9
Table 3-6. Microtec Preprocessor Directives 3-11
Table 4-1. C Library Customizer Preprocessor Symbols 4-4
Table 5-1. mriext.h Functions 5-1
Table 5-2. mriext.h Macros 5-2
Table 5-3. Non-Reentrant Functions and Extensions 5-3
Table 5-4. System Functions 5-6
Table 5-5. Mode Values for the open Function 5-26
Table 6-1. Parameter Limits 6-3
Table 6-2. Scalar Data Types 6-4
Table 6-3. Complex (Aggregate) Types 6-5
Table 6-4. Mixed Operand Conversion 6-6
Table 6-5. Integral Promotion 6-6
Table 6-6. Natural Boundary Alignment for the 68000 Family 6-19
Table 6-7. Effect of Compiler Options on Structure Layout 6-22
Table 7-1. Instructions for Removing Bytes from Stack 7-7
Table 8-1. Sections Generated by the Compiler 8-1
Table 8-2. Example Modules 8-3
Table 8-3. Sections Contained in module1.o and module2.o 8-3
Table 8-4. Sections Contained in the Executable File 8-4
Table 9-1. Position-Independent Code Options 9-21
Table 9-2. Absolute Versus Register-Relative Addressing 9-27
Table 9-3. Commands for Absolute Addressing Example 9-29
Table 9-4. Commands for PC-Relative Addressing Example 9-30
Table A-1. Message Severity Levels A-2
Table A-2. Using Compiler Options to Suppress Diagnostic Messages A-4

Index

Symbols

_ _DATE_ _ preprocessor symbol 3-2
_ _FILE_ _ preprocessor symbol 3-2
_ _LINE_ _ preprocessor symbol 3-2
_ _OPTION_VALUE preprocessor symbol 3-9
_ _STDC_ _ preprocessor symbol 2-16, 2-51, 3-4, 3-7
_ _STR_CASE_CMP preprocessor symbol 3-9
_ _STR_CMP preprocessor symbol 3-9
_ _STRIP_QUOTES preprocessor symbol 3-9
_ _TIME_ _ preprocessor symbol 3-4
16-bit bus 6-18
16-bit displacement
    (see also -Ml option) 2-31
16-bit extension on stack
    -Zp2 option 2-29
32-bit displacement
    -Ml option 2-31
32-bit extension on stack
    -Zp4 option 2-29
_68000 preprocessor symbol 3-7
_68008 preprocessor symbol 3-7
_68010 preprocessor symbol 3-7
_68020 preprocessor symbol 3-7
_68030 preprocessor symbol 3-7
_68040 preprocessor symbol 3-8
_68060 preprocessor symbol 3-8
_68302 preprocessor symbol 3-8
_68330 preprocessor symbol 3-8
_68331 preprocessor symbol 3-8
_68332 preprocessor symbol 3-8
_68333 preprocessor symbol 3-8
_68340 preprocessor symbol 3-8
_68349 preprocessor symbol 3-8
_68360 preprocessor symbol 3-8
_68EC000 preprocessor symbol 3-8
_68EC020 preprocessor symbol 3-8
_68EC030 preprocessor symbol 3-8
_68EC040 preprocessor symbol 3-8
_68EC060 preprocessor symbol 3-8
_68HC000 preprocessor symbol 3-8
_68HC001 preprocessor symbol 3-8

A

-A option 2-16, 3-4, 3-7
Absolute addressing 9-27
    -Mca option 2-30
    -Mda option 2-30
-acc option 2-17
-acd option 2-17
acos function 2-47
Address modes
    code references
        absolute
            -Mca option 2-30
        PC-relative
            (see also Position-independent code)
            -Mcp option 2-30
    const section
        -ac options 2-17
    data references
        absolute
            -Mda option 2-30
        PC-relative
            (see also Position-independent data)
            -Mdp option 2-30
        register-relative
            (see also Position-independent data)
            -Md options 2-30
    initialized data section
        -ai options 2-17
    literals section
        -al options 2-17
Addressing
    absolute 9-27
    changing the default 8-2

PC-relative 9-29
register-relative 9-27
Addressing of processors
odd-address restricted 6-13
odd-address unrestricted 6-13
Aggregates 6-16
-aic option 2-17
-aid option 2-17
-alc option 2-17
-ald option 2-17
Algebraic simplification 10-1
alias command (Linker) 2-33
Aliased references
-Ob option 2-34
Alignment
aggregates 6-16
array 6-16
bit fields 6-19
data types 6-19
padding bytes 6-20
problems
-KT option 2-28
structure members
-Z options 2-47
structures 6-16, 6-20
trailer bytes 6-20
unions 6-16
Alignment of struct/union in parameter area 7-4
Allocating
data space 5-38
data types 6-3
memory space 5-31
Alternate locations
UNIX
executables 2-4
libraries 2-4
standard include files 2-4
temporary files 2-4
ANSI-compliant mode, setting
-A option 2-16
Array operator synthesis optimization 10-7
Arrays 6-5
-asc option 2-17
ASCII character, testing for 5-16
ASCII format
conversion from byte 5-33
ASCII string
conversion from floating-point number 5-13
conversion from integer 5-17
conversion from long integer 5-20
conversion from unsigned integer 5-18
conversion from unsigned long integer 5-21
-asd option 2-17
asin function 2-47
asm pseudofunction 9-3
enabling
-x option 2-46
Assembler
description 1-2
options, passing directly
-Wa option 2-10, 2-14, 2-44
source file
(see Assembly source file)
Assembler in-lining 9-3
considerations 9-9
Assembler source file 11-1
advantages to producing 11-1
contents 11-3
line numbers 11-2
variable names 11-1
Assembly and high-level code
-Fsm option 2-20
Assembly code, optimizing by hand 11-1
Assembly file
saving
-H option 2-23
Assembly language
example of a routine 7-10
interface 7-1, 7-9
Assembly source file
generating
-S option 2-42

high-level source code, including as comments
-Fsm option 2-20
#include files expanded
-Fsi option 2-20
naming
-o option 2-38
atan function 2-47
atanh function 2-47

B

Backspace
disable for preprocessor
-Es option 2-19
-Ps option 2-38
Bandwidth, data bus 6-14, 6-18
Banner
-Vb option 2-44
BIG_ENDIAN 6-1
_BIG_ENDIAN preprocessor symbol 3-2
Bit fields 6-11
alignment 6-19
packing 6-13
bld_lib script 4-7
Branch tail merging 10-12
Bus bandwidth 6-14, 6-18
Bytes
clearing memory bytes 5-24
conversion to ASCII format 5-33
reading from a file 5-30
swapping odd and even bytes 5-32
writing to a file 5-37

C

C calling conventions 7-1
C comments, saving in preprocessor output
-C option 2-17
C common 2-45
-C option 2-17
-c option 2-18
example use 2-49
C++ comments
-zc 2-48
C++ compiler
description 1-2
C++ compiler package
C++ compiler 1-2
C++ exception handling
-ze option 2-48
calloc function 9-12
Change search path order
-I@ option 2-24
char type 6-4
Character
converting to lowercase 5-34
converting to uppercase 5-35
copying characters from memory 5-23
testing for ASCII character 5-16
_CHAR_SIGNED preprocessor symbol 2-28, 3-7
_CHAR_UNSIGNED preprocessor symbol 2-28, 3-7
Checking
extra
-v option 2-44
syntax only
-y option 2-47
Clearing memory bytes 5-24
close function 2-64, 5-8, 9-33
Closing a file 5-8
Code
position-dependent 9-21
position-independent 9-21
Code and data section names, specifying 2-31
Code elimination, unreachable 10-2
Code hoisting 10-12
Code organization 8-1
compiler-generated sections 8-3
Code references
absolute
-Mca option 2-30
PC-relative
-Mcp option 2-30
code section 2-16, 8-4

naming
-NT option 2-32
Column number debugging information
-Gm option 2-22
Command file, passing to linker
-e option 2-19
Command line options
DOS 2-1
specifying in file
-d option 2-19
UNIX 2-1
Comments, saving in preprocessor output
-C option 2-17
Comparing values 5-22, 5-25
Compiler
command line 2-1
description 1-1
input file extensions 2-2
invoking 2-1
syntax 2-1
Compiler features 1-1
Compiler options
UNIX/DOS
-A 3-4, 3-7
-e 9-25
-g 3-7
-Gl 11-2
-Kf 10-17
-Kh 9-36
-Kr 9-46
-Ks 9-19
-Ku 3-7, 6-4
-Mcp 9-21, 9-22
-Md 3-7
-Mdn 9-21, 9-22
-Mdp 9-21, 9-22
-nKu 3-7
-nOc 10-18
-Og 10-12
-Os 10-1
-Ot 10-1
-p 3-7
-Q A-4
-Qe A-4
-Qi A-4
-Qs A-4
-Qw A-4
-utn 3-7
-v A-20
-x 3-7
-Zp2 6-8
-Zp4 6-8
Compiler output
assembler source file 11-1
listing 11-3
Compiler-generated literals section 2-16
naming
-NL option 2-32
Compiler-generated tag data section 2-16
-Kt option 2-28
Complex data types 6-5
Complex straddling 6-14
Configuration file 4-3
Configurations, memory 9-17, 9-45
Conflicting options, specifying 2-6
const section 8-5
naming
-NC option 2-31
specifying address mode
-ac options 2-17
Constant folding optimization 10-10
Constant literals section
-al options 2-17
Constant variables section
naming
-NC option 2-31
specifying address mode
-ac options 2-17
Constants
floating-point size
-Kq option 2-27
Conventions, notational viii
Copying characters from memory 5-23
cos function 2-47
cosh function 2-47
_CPU32 preprocessor symbol 3-8

_CPU32P preprocessor symbol 3-8
creat function 2-64, 5-9
    relationship to close 5-8
    relationship to read 5-30
    relationship to write 5-37
Creating a file 5-9

D

-D option 2-18
-d option 2-19
Data
    formats 6-1
    initialization 9-39
    position-dependent 9-21
    position-independent 9-21
    system 9-37
    types 6-3
        allocation 6-3
        ranges 6-3
Data browser
    -GS option 2-22
Data bus bandwidth 6-14, 6-18
Data references
    absolute
        -Mda option 2-30
    PC-relative
        -Mdp option 2-30
    register-relative
        -Md options 2-30
Data space
    allocating 5-38
Data type alignment 6-19
_ _DATE_ _ preprocessor symbol 3-2
Dead code elimination 10-2
_DEBUG preprocessor symbol 2-22, 3-7
Debugging information
    fully qualified path names
        -Gf option 2-21
    generating
        -g option 2-22
    line number information
        -Gl option 2-21
    multiple statements on line1opb
        -Gm option 2-22
    preprocessor macros
        -Gd option 2-21
    XRAY Source Explorer
        -GS option 2-22
        -Gs option 2-22
Default command, example use and description 2-49
Default initialization 6-27
#define directive 9-6
Defining macros on command line
    -D option 2-18
Diagnostic messages
    (see also Messages, diagnostic)
    suppressing
        -Q option 2-41
Disable
    backspace
        -Es option 2-19
        -Ps option 2-38
    newline
        -Es option 2-19
        -Ps option 2-38
    optimizations on global variables
        -Ob option 2-34
    stack frame sharing
        -Kf option 2-25
Display time stamp
    -Vt option 2-44
DOS
    command line 2-1
    compiler syntax 2-1
    compiler use 2-1
    compiling a program 2-65
    environment variables 2-3
    file locations 2-3
    file name defaults 2-2
    invoking the compiler 2-1
    libraries provided 2-60
    option form, positive and negative 2-6
    option, command line
        descriptions of 2-6

double type 6-4
Driver 2-1

E

-E option 2-19
-e option 2-19, 9-25
Embedded systems
  code organization 8-1
  considerations 9-1
  user-modified routines 9-31
enum type 6-4
Enumerated data types
  packing 6-25
Enumerator types, packed
  -D option 2-18
Environment variables 2-3
  DOS
    MRI_68K_BIN 2-4
    MRI_68K_INC 2-4
    MRI_68K_LIB 2-4
    MRI_68K_TMP 2-4
    USR_MRI 2-4
  UNIX
    MRI_68K_BIN 2-4
    MRI_68K_INC 2-4
    MRI_68K_LIB 2-4
    MRI_68K_TMP 2-4
    USR_MRI 2-4
Epilogue 10-17
  function
    generating code for epilogue 10-17
Epilogue, function 7-8
eprintf function 5-3, 5-10
#error directive
  relation to #pragma error 3-16
Error messages A-1
  suppression
    -Qe option 2-41
-Es option 2-19
Exception handling 2-54
Exception handling, C++
  -ze option 2-48
EXCLUDE_AT_EXIT preprocessor symbol 4-6
EXCLUDE_ERRNO preprocessor symbol 4-6
EXCLUDE_FORMAT_IO_ASSGN_SUPP
  preprocessor symbol 4-4
EXCLUDE_FORMAT_IO_BRAKT_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_CHAR_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_DEC_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_FLOAT_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_HEX_FMT
  preprocessor symbol 4-6
EXCLUDE_FORMAT_IO_h_OPT
  preprocessor symbo 4-4
EXCLUDE_FORMAT_IO_INT_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_L_OPT
  preprocessor symbol 4-4
EXCLUDE_FORMAT_IO_l_OPT
  preprocessor symbol 4-4
EXCLUDE_FORMAT_IO_MINUS_FLAG
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_NUMB_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_OCT_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_PLUS_FLAG
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_PNT_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_SHARP_FLAG
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_SPACE_FLAG
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_STAR_OPT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_STR_FMT
  preprocessor symbol 4-5
EXCLUDE_FORMAT_IO_UNS_FMT
  preprocessor symbol 4-6

EXCLUDE_FORMAT_IO_ZERO_FLAG preprocessor symbol 4-5
EXCLUDE_INITDATA preprocessor symbol 4-6
EXCLUDE_IOB preprocessor symbol 4-6
EXCLUDE_LINE_BUFFER_DEFAULT preprocessor symbol 4-6
EXCLUDE_RAND preprocessor symbol 4-6
EXCLUDE_SIGNAL_RAISE preprocessor symbol 4-6
EXCLUDE_STRTOK preprocessor symbol 4-6
EXCLUDE_TERMINAL_SIMULATION preprocessor symbol 4-6
EXCLUDE_TIME preprocessor symbol 4-6
Executable file
  suppressing
    -c option 2-18
Executable program
  generation 2-65
  running on a target system 2-65
Executables
  UNIX
    alternate locations 2-4
Executing only preprocessor
  -E option 2-19
  -P option 2-38
_exit function 2-64, 5-11
exit function
  relationship to _exit 5-11
exp function 2-47

Explorer, generating code for
  -GS option 2-22
  -Gs option 2-22
Extensions
  file name 2-2
  Microtec
    (see Microtec extensions)
Extensions to C
  -x option 2-46
External variable names 11-1

F

-f option 2-21
fabs function 2-47
Features of compiler 1-1
-Fee option 2-19
-Feo option 2-20
File
  closure 5-8
  creation 5-9
  current location 5-19
  making file inaccessible 5-36
  opening 5-26
  unlinking a file name 5-36
File descriptor, getting 5-12
_ _FILE_ _ preprocessor symbol 3-2
fileno macro 5-12
Files
  #include
    (see #include files)
  input
    extensions 2-2
    locations 2-3
  linker command 2-19
  listing
    (see Listing files)
  output
    extensions 2-3
    locations 2-3
-Fli option 2-20
float type 6-4
Floating-point
  number
    conversion to ASCII string 5-13
  removing unneeded support 9-35
  return values 7-5
Floating-point constant size
  -Kq option 2-27
Floating-point coprocessor instructions, generating
  -f option 2-21
-Flp option 2-20
-Flp0 option 2-20
-Flt option 2-20

Formatted output to standard error 5-10
Frame pointer 7-6
free function 9-12
-Fsm option 2-20
ftell function 2-64
ftoa function 5-13
Fully qualified names, generating for input files
    -Gf option 2-21
Function declaration
    with interrupt keyword 2-54, 6-8
Function in-lining 10-15
Function names, truncating
    -ut option 2-43
Functions 6-7
    calling at an absolute address 9-10
    epilogue 7-8
    library
        (see Library functions)
    non-reentrant 5-3
    prologue 7-7
        local variables in prologue 7-8
    return values
        floating-point 7-5
        integer 7-5
        structure 7-5
        union 7-5
    returning from
        -Oe option 2-35
    system 9-33
    tagging entry and exit points
        -Kt option 2-28

G

-g option 2-22, 3-7
General optimizations 10-1
Generating floating-point processor
        instructions
    -f option 2-21
getl function 5-14
getw function 5-4, 5-15
-Gf option 2-21
-Gl option 2-21, 11-2
Global constant propagation optimization 10-4
Global copy propagation 10-5
Global optimizations 10-2 to 10-9
Global value propagation 10-5
Global variable optimization
    -Ob option 2-34
Global variables
    uninitialized 2-45
Global-flow optimizer
    -Og option 2-35
-Gm option 2-22
Grouping stack adjust instructions optimization
        10-18
-GS option 2-22
-Gs option 2-22

H

-H option 2-23
-h option 2-23
_H7U preprocessor symbol 3-6
High-level and assembly code
    -Fsm option 2-20
HP 64000 Systems, generating code for
    -h option 2-23
_HP9000_700 preprocessor symbol 3-6

I

-I option 2-23
-I@ option 2-24
Include files
    UNIX
        alternate locations 2-4
#include files
    expanding in assembler source file
        -Fsi option 2-20
    mriext.h 5-1, 5-2
    search path, specifying
        -I option 2-23
        -J option 2-25
    setjmp.h 5-7
INCLUDE_BUILD_ARGV preprocessor
        symbol 4-6

INCLUDE_FAST_POW preprocessor symbol 4-7
Index simplification optimization 10-9
Indexing array optimization 10-18
Induction variable elimination 10-9
#info directive 3-12
  relation to #pragma info 3-17
Informational messages
  suppression
    -Qi option 2-41
INITDATA linker command 9-41
initfini section 2-16
Initialization 9-40
  compile-time 9-39
  data 9-39
  default 6-27
  run-time 9-39
  static variables 6-27
Initialize local variables
  -KI option 2-26
Initialized data section 2-16
  naming
    -NI option 2-31, 2-32
  specifying address mode
    -ai options 2-17
initvars section 8-5
In-line assembly 9-2, 9-3
In-line assembly code
  changing insert character
    -ui option 2-43
  enabling
    -x option 2-46
In-line library function expansion 10-17
In-lining
  -Oi option 2-35
In-lining run-time library functions
  -Oj option 2-36
Input and output files, location
  DOS 2-3
  UNIX 2-3
Input files
  extensions 2-2
  generating fully qualified names
    -Gf option 2-21
  locations 2-3
Insert character, changing
  -ui option 2-43
int type 6-4
Integer
  conversion to ASCII string 5-17
Integer return values 7-5
Internal compiler errors A-16
Interrupt handlers, declaring 7-12, 9-46
interrupt keyword 2-54, 6-8, 7-12, 9-46
Interrupt procedures
  -Kr option 2-27
  return with RTE instruction
    -nKr option 2-27
  return with RTS instruction
    -Kr option 2-27
Introduction to compiler package 1-1
Invocation
  compiler 2-1
  linker 2-66
I/O device registers 2-34
ioports section 8-5
isascii function 5-16
itoa function 5-17
itostr function 5-18

J

-J option 2-25
Jump optimizations 10-11 to 10-15

K

Keywords
  interrupt 2-54, 6-8
  packed 2-54
  typeof 2-54
  unpacked 2-54
-Kf option 2-25, 7-7, 10-17
-Kh option 2-26, 7-6, 9-36
-KI option 2-26
-Km option 2-26

-Kq option 2-27
-Kr option 2-27, 7-12, 9-46
-Ks option 9-19
-KT option 2-28
-Kt option 2-28
-Ku option 2-28, 3-7, 6-4

L

-l option 2-30
Labels for line numbers
    -Gl option 2-21
Librarian description 1-3
Libraries 2-60, 9-40
    library customizer 4-1
    UNIX
        alternate locations 2-4
Library customizer 4-1
    building custom libraries 4-7
    configuration file 4-3
    directories 4-2
    preprocessor symbols 4-4
    testing custom libraries 4-8
Library functions
    (see under specific names)
    close 9-33
    lseek 9-33
    non-reentrant 5-3
    open 9-33
    read 9-33
    write 9-33
Library macros 5-2
#line directive 3-2
Line number labels, producing
    -Gl option 2-21
Line number, variable, and symbol information
    -g option 2-22
Line numbers in assembly language source file 11-2
_ _LINE_ _ preprocessor symbol 3-2
LINE_BUFFER_SIZE preprocessor symbol 4-6
Linker
    alias command 2-33
    command example
        IEEE 9-42
        ROM-based system, IEEE 9-44
    command file
        alternate 2-19
        default 2-19
        passing
            -e option 2-19
    description 1-2
    invoking 2-66
    passing default libraries 2-42
    passing options directly
        -Wl option 2-45
    suppressing call to 2-18
Linker command files
    use 9-40
Listing files
    format
        -Fli option 2-20
    generating
        -l option 2-30
    omitting page header
        -Flp0 option 2-20
    page length, specifying
        -Flp option 2-20
    title
        -Flt option 2-20
literals section 2-16, 8-5
    naming
        -NL option 2-32
    specifying address mode
        -al options 2-17
Little 6-1
_LITTLE_ENDIAN preprocessor symbol 3-2
Local optimizations 10-10
    -Ol option 2-36
Local variable initialization
    -KI option 2-26
Local variables 6-27
Location
    current in a file 5-19
Locations of input and output files
    DOS 2-3

UNIX 2-3
log function 2-47
log10 function 2-47
long double type 6-4
Long integer
conversion to ASCII string 5-20
reading from a stream 5-14
writing to stream 5-28
long type 6-4
Loop optimizations 10-6
Lower-case characters
converting to 5-34
lseek function 2-64, 5-19, 9-33
ltoa function 5-20
ltostr function 5-21

M

Machine-dependent optimizations 10-17 to 10-19
Macros
defining on command line
-D option 2-18
undefining
-U option 2-42
malloc function 9-12
MasterWorks
(see XRAY MasterWorks)
math.h include file 9-12
max macro 5-22
-Mca option 2-30
mcc68k020a5.lib 2-62
mcc68k020ab.lib 2-61
mcc68k020pc.lib 2-62
mcc68k040a5.lib 2-62
mcc68k040ab.lib 2-62
mcc68k040pc.lib 2-62
mcc68k32a5.lib 2-61
mcc68k32ab.lib 2-61
mcc68k32pc.lib 2-61
mcc68ka5.lib 2-61
mcc68ka5020f.lib 2-62
mcc68ka532f.lib 2-61
mcc68ka5f.lib 2-61
mcc68kab.lib 2-61
mcc68kab020f.lib 2-62
mcc68kab32f.lib 2-61
mcc68kabf.lib 2-61
mcc68kpc.lib 2-61
mcc68kpc020f.lib 2-62
mcc68kpc32f.lib 2-61
mcc68kpcf.lib 2-61
-Mcp option 2-30, 9-21, 9-22
-Md option 2-30, 3-7
-Mda option 2-30
-Mdn option 9-21, 9-22
-Mdp option 2-30, 9-21, 9-22
memccpy function 5-23
memclr function 5-24
memcpy function 2-47
Memory configurations 9-17, 9-45
Memory layout 6-27
Memory space
allocating 5-31
Message severity levels A-2
Messages, diagnostic
displaying
-nQ option 2-41
suppressing
-Q option 2-41
writing to stderr
-Fee option 2-19
writing to stdout
-Feo option 2-20
Messages, error A-1
Messenger symbols 9-41
-Kir option 2-26
Microprocessor
(see Processor, specifying)
Microtec extensions 2-46
functions 5-1
eprintf 5-10
ftoa 5-13
getl 5-14
getw 5-15
isascii 5-16
itoa 5-17

itostr 5-18
ltoa 5-20
ltostr 5-21
memccpy 5-23
memclr 5-24
putl 5-28
putw 5-29
swab 5-32
toascii 5-33
_tolower 5-34
_toupper 5-35
zalloc 5-38
macros
BLKSIZE 5-2
FALSE 5-2
fileno 5-12
isascii 5-16
max 5-22
min 5-25
NULLPTR 5-2
stdaux 5-2
stdprn 5-2
toascii 5-33
_tolower 5-34
_toupper 5-35
TRUE 5-2
-x option 2-46
_MICROTEC preprocessor symbol 3-3
min macro 5-25
Mixed operands 6-5
-Ml option
generate position-independent code and data 2-31
Modes
address
(see Address modes)
ANSI-compliant
-A option 2-16
processor
-p option 2-38
verbose
-V options 2-44
Module, naming
-NM option 2-32
_MRI preprocessor symbol 3-2
MRI_68K_BIN 2-4
MRI_68K_INC 2-4
MRI_68K_LIB 2-4
MRI_68K_TMP 2-4
mriext.h include file 5-1, 5-2
_MRI_EXTENSIONS preprocessor symbol 2-46, 3-7
_MSD preprocessor symbol 3-6
_MSDOS preprocessor symbol 3-6
Multiple jump optimization 10-14
Multitasking environment (MTE) 9-13, 9-35
Multi-threaded environments 9-13

N

-nA option 2-16
Names, truncating
-ut option 2-43
Naming convention for symbols 2-43
Naming modules
-NM option 2-32
-NC option 2-31
-nC option 2-17
-nc option 2-18
-nE option 2-19
Negative option prefix 2-6
Newline
disable for preprocessor
-Es option 2-19
-Ps option 2-38
-nf option 2-21
-nFee option 2-19
-nFeo option 2-20
-nFli option 2-20
-nFsi option 2-20
-nFsm option 2-20
-ng option 2-22
-nGf option 2-21
-nGl option 2-21
-nGm option 2-22
-nGr option 2-22

-nGS option 2-22
-nGs option 2-22
-nH option 2-23
-nh option 2-23
-NI option 2-31
-nKc option 2-25
-nKf option 2-25
-nKI option 2-26
-nKm option 2-26
-nKP option 2-27
-nKq option 2-27
-nKr option 2-27
-nKT option 2-28
-nKt option 2-28
-nKu option 2-28, 3-7
-NL option 2-32
-nl option 2-30
-NM option 2-32
-nO option 2-34
-nOb option 2-34
-nOc option 2-35, 7-7, 10-18
-nOe option 2-35
-nOg option 2-35
-nOi option 2-35
-nOj option 2-36
-nOl option 2-36
Non-reentrant functions
(see also Reentrant functions)
-nOR option 2-36
-nOr option 2-37
Notational conventions viii
-nP option 2-38
-nQ option 2-41
-nq option 2-42
-nQo option 2-41
-NS option 2-32
-NT option 2-32
-nV option 2-44
-nv option 2-44
-nx option 2-46
-ny option 2-47
-NZ option 2-32
-nzc option 2-48
-nZe option 2-29

O

-O option 2-34
-o option 2-38
-Ob option 2-34
Object files
naming
-o option 2-38
producing only
-c option 2-18
-Oc option 2-35
Odd-address restricted processors 6-13
Odd-address unrestricted processors 6-13
-Oe option 2-35
-Og option 2-35, 10-12
-Oi option 2-35
-Oj option 2-36
-Ol option 2-36
open function 2-64, 5-26, 9-33
relationship to close 5-8
relationship to read 5-30
relationship to write 5-37
Opening a file 5-26
Optimizations 10-1
algebraic simplification 10-1
array operator synthesis 10-7
branch tail merging 10-12
code for epilogue 10-17
code for prologue 10-17
constant folding 10-10
cross-jump 10-13
general 10-1
global 10-2 to 10-9
global constant propagation 10-4
global copy propagation 10-5
grouping stack adjust instructions 10-18
indexing arrays 10-18
in-line library function expansion 10-17
instruction scheduling 10-17
jump 10-11 to 10-15
local 10-10
loop 10-6

machine dependent 10-17 to 10-19
multiple jump 10-14
redundant code elimination 10-2
redundant jump elimination 10-14
short/long displacement 10-15
strength reduction 10-2
strength reduction and index simplification 10-9
Optimizing code 2-34
execution time
-Ot option 2-37
global-flow optimizer
-Og option 2-35
in-lining
-Oi option 2-35
local optimizations
-Ol option 2-36
size
-Os option 2-37
Option descriptions
DOS 2-6
UNIX 2-6
Options
active on output listing
suppression
-Qo option 2-41
command line 2-6
conflicting, specifying 2-6
descriptions 2-15
file, specifying
-d option 2-19
negative 2-6
passing to assembler
-Wa option 2-10, 2-14, 2-44
passing to linker
-Wl option 2-45
summary 2-6
_OPTION_utn preprocessor symbol 3-7
_ _OPTION_VALUE preprocessor symbol 3-9
-Os option 2-37, 10-1
-Ot option 2-37, 10-1
Output
assembler source file 11-1
listing 11-3
Output files
extensions 2-3
locations 2-3
naming
-o option 2-38
specifying format
-Fli option 2-20

P

-P option 2-38
-p option 2-38, 3-7
Packed enumerator type
-D option 2-18
packed keyword 2-54, 6-17
Packed structures 6-17, 6-19
-D option 2-18
tips 6-24
packed type 6-23
Packed unions and structures
-KP option 2-27
_PACKED_STRUCTS preprocessor symbol 3-3
Packing
bit fields 6-13
enumerated data types 6-25
tips 6-24
Padding 6-8
bytes 6-20
structures 6-18
Page header, specifying for listing file
-Flp0 option 2-20
Parameters
passing 7-1
popping 7-5
setting 7-1
Passing options
to assembler
-Wa option 2-10, 2-14, 2-44
to linker
-Wl option 2-45

_PC preprocessor symbol 3-6
PC-relative address mode
  code references
    (see also Position-independent code)
    -Mcp option 2-30
  data references
    (see also Position-independent data)
    -Mdp option 2-30
PC-relative addressing 9-29
_PIC preprocessor symbol 2-30
_PID preprocessor symbol 2-30, 3-7
_PID_REG preprocessor symbol 2-30, 3-7
Pipeline instruction scheduling 10-17
pixinit section 2-16
Pointers 6-5
  types 6-4
Popping stack
  -Oc option 2-35
Position-dependent
  code 9-21
  data 9-21
Position-independent
  code 9-21
  data 9-21
Position-independent code
  -Mcp option 2-30
Position-independent data
  -Md options 2-30
#pragma asm directive 3-13, 3-15, 9-7
#pragma eject directive 3-14
#pragma endasm directive 9-8
#pragma error directive 3-16
#pragma info directive 3-17
#pragma list directive 3-18
#pragma macro directive 3-19
#pragma option directive 3-20, 9-9
#pragma options 6-22
#pragma warn directive 3-21
Predefined symbols 3-1
  _ _DATE_ _ 3-2
  _ _FILE_ _ 3-2
  _ _LINE_ _ 3-2
  _ _STDC_ _ 3-4
  _ _TIME_ _ 3-4
  _ _OPTION_VALUE 3-9
  _ STDC_ _ 3-7
  _ _STR_CASE_CMP 3-9
  _ _STR_CMP 3-9
  _ _STRIP_QUOTES 3-9
  _68000 3-7
  _68008 3-7
  _68010 3-7
  _68020 3-7
  _68030 3-7
  _68040 3-8
  _68060 3-8
  _68302 3-8
  _68330 3-8
  _68331 3-8
  _68332 3-8
  _68333 3-8
  _68340 3-8
  _68349 3-8
  _68360 3-8
  _68EC000 3-8
  _68EC020 3-8
  _68EC030 3-8
  _68EC040 3-8
  _68EC060 3-8
  _68HC000 3-8
  _68HC001 3-8
  _BIG_ENDIAN 3-2
  _CHAR_SIGNED 3-7
  _CHAR_UNSIGNED 3-7
  _CPU32 3-8
  _CPU32P 3-8
  _DEBUG 3-7
  H7U 3-6
  _HP9000_700 3-6
  _LITTLE_ENDIAN 3-2
  _MICROTEC 3-3
  _MRI 3-2
  _MRI_EXTENSIONS 3-7
  _MSD 3-6
  _MSDOS 3-6
  _OPTION_utn 3-7

_PACKED_STRUCTS 3-3
_PC 3-6
_PID 3-7
_PID_REG 3-7
_PTRDIFF_T 3-3
_S4U 3-6
_SIZEOF_CHAR 3-3
_SIZEOF_CODE_POINTER 3-3
_SIZEOF_DATA_POINTER 3-3
_SIZEOF_DOUBLE 3-3
_SIZEOF_FLOAT 3-3
_SIZEOF_INT 3-3
_SIZEOF_LONG 3-3
_SIZEOF_LONG_DOUBLE 3-4
_SIZEOF_POINTER 3-4
_SIZEOF_SHORT 3-4
_SIZE_T 3-3
_STACK_GROWS_DOWN 3-4
_STACK_GROWS_UP 3-4
_STR_CASE_CMP 3-9
_STR_CMP 3-9
_SUN4 3-6
_UNIX 3-6
_VERSION 3-4
_WCHAR_T 3-4
Preprocessor 3-1
executing only
-E option 2-19
-P option 2-38
macros
defining on command line
-D option 2-18
undefining
-U option 2-42
output
saving comments
-C option 2-17
sending to standard output
-E option 2-19
Preprocessor directives
#info 3-12
#pragma asm 3-13, 3-15
#pragma eject 3-14
#pragma error 3-16
#pragma info 3-17
#pragma list 3-18
#pragma macro 3-19
#pragma option 3-20
#pragma warn 3-21
#warning 3-22
preprocessor symbol 4-6
Preprocessor symbols 4-4
printf function
preprocessor symbols 4-4
removing unneeded I/O support 9-34
Processor, specifying
-p option 2-38
Program identifiers, truncating
-ut option 2-43
Program termination 5-11
Prologue 10-17
function
generating code for prologue 10-17
Prologue, function 7-7
-Ps option 2-38
_PTRDIFF_T preprocessor symbol 3-3
Public
variable names 11-1
putl function 5-5, 5-28
relationship to getl 5-14
putw function 5-5, 5-29
relationship to getw 5-15

Q

-Q option 2-41, A-4
-q option 2-42
-Qe option 2-41, A-4
-Qi option 2-41, A-4
-Qo option 2-41
-Qs option 2-41, A-4
Questions viii
-Qw option 2-41, A-4

R

Range information
    producing during compilation 2-54
read function 2-64, 5-30, 9-33
Reading bytes from a file 5-30
realloc function 9-12
Redundant code elimination 10-2
Redundant jump optimization 10-14
Redundant store elimination 10-11
Reentrant code
    generating 9-11
Reentrant functions 5-3, 9-11
Register
    reserving a register 9-35
    reserving for special purposes 9-38
    storage class 6-26
    use 7-6
register
    keyword 6-26
Register storage class 6-26
Register-relative address mode 2-30
    (see also Position-independent data)
Register-relative addressing 9-27
Registers
    I/O device 2-34
    reserving
        -Kh option 2-26
Reserving a register
    -Kh option 2-26
Returning a typed value 9-5
ROM-based systems 9-44
Routines, user-modified 9-31
RTE instruction
    using for interrupt procedures
        -nKr option 2-27
RTS instruction
    using for interrupt procedures
        -Kr option 2-27

S

-S option 2-42
_S4U preprocessor symbol 3-6
sbrk function 2-64, 5-31
scanf function
    preprocessor symbols 4-4
Search paths
    (see also Environment variables)
    nonstandard #include files
        -I option 2-23
    standard #include files
        -J option 2-25
Section names, specifying 2-31
setjmp.h include file 5-7
Setting sign of char variables
    -Ku option 2-28
Severity of errors A-2
Shared
    program 9-36
Short integer
    reading from a stream 5-15
    writing to stream 5-29
short type 6-4
Short/long displacement optimizations 10-15
Simple straddling 6-14
_simulated_input variable 9-33
_simulated_output variable 9-33
sin function 2-47
Single-precision math, enabling 2-27
sinh function 2-47
Size
    aggregates 6-16
    data types 6-3
    unpacked structure 6-21
Size optimization
    -Os option 2-37
_SIZEOF_CHAR preprocessor symbol 3-3
_SIZEOF_DOUBLE preprocessor symbol 3-3
_SIZEOF_FLOAT preprocessor symbol 3-3
_SIZEOF_INT preprocessor symbol 3-3
_SIZEOF_LONG_DOUBLE preprocessor
        symbol 3-4
_SIZEOF_LONG preprocessor symbol 3-3
_SIZEOF_POINTER preprocessor symbol 3-4
_SIZEOF_SHORT preprocessor symbol 3-4
_SIZEOF_CHAR preprocessor symbol 3-3

_SIZEOF_CODE_POINTER preprocessor symbol 3-3
_SIZEOF_DATA_POINTER preprocessor symbol 3-3
_SIZEOF_DOUBLE preprocessor symbol 3-3
_SIZEOF_FLOAT preprocessor symbol 3-3
_SIZEOF_INT preprocessor symbol 3-3
_SIZEOF_LONG preprocessor symbol 3-3
_SIZEOF_LONG_DOUBLE preprocessor symbol 3-4
_SIZEOF_POINTER preprocessor symbol 3-4
_SIZEOF_SHORT preprocessor symbol 3-4
_SIZE_T preprocessor symbol 3-3
size_t type
  memccpy function 5-23
  memclr function 5-24
  zalloc function 5-38
Source file problems, detecting
  -v option 2-44
sprintf function 9-11
sqrt function 2-47
sscanf function 9-11
Stack
  16-bit quantities
    -Zp2 option 2-29
  32-bit quantities
    -Zp4 option 2-29
  disabling frame sharing
    -Kf option 2-25
  frames 7-6
  int size
    -Ze option 2-29
  pointer 7-6
  popping
    -Oc option 2-35
_STACK_GROWS_DOWN preprocessor symbol 3-4
_STACK_GROWS_UP preprocessor symbol 3-4
Start-up routine 2-64
Static
  variables 6-27
    names 11-2
_ _STDC_ _ preprocessor symbol 3-4
_ _STDC_ _ preprocessor symbol 2-16, 2-51
stderr
  directing diagnostic messages
    -Fee option 2-19
stdio.h include file 9-11
stdout
  directing diagnostic messages
    -Feo option 2-20
Storage
  allocation of data types 6-3
  classes
    keywords 6-26
  layout 6-1
Straddling bits 6-14
_STR_CASE_CMP preprocessor symbol 3-9
_ _STR_CASE_CMP preprocessor symbol 3-9
_STR_CMP preprocessor symbol 3-9
_ _STR_CMP preprocessor symbol 3-9
strcpy function 2-47
Stream
  reading a long integer 5-14
  reading a short integer 5-15
  writing a long integer 5-28
  writing a short integer 5-29
Strength reduction 10-2
  optimization 10-9
strings section 2-16, 8-4
  naming
    -NS option 2-32
  specifying address mode
    -as options 2-17
strlen function 2-47
strtod function 9-12
strtol function 9-12
strtoul function 9-12
struct type 6-5
struct/union in parameter area, alignment of 7-4
Structure
  return value 7-5
Structure alignment 6-20

St1ructure members, aligning
-Z options 2-47
Structure padding 6-18
Structure size
-Zm option 2-48
Structures 6-8
packed 6-17
packing 6-24
size 6-21
Structures, packed
-D option 2-18
Suggestions viii
Summary message
suppression
-Qs option 2-41
Summary of command line options 2-6
_SUN4 preprocessor symbol 3-6
Superscalar instruction scheduling 10-17
Suppressing executable file
-c option 2-18
swab function 5-32
Swapping bytes in memory 5-32
switch statement
generating code for 10-10
Symbol names
convention 2-43
prepend dot
-upd option 2-43
prepend underscore
-upu option 2-43
suppress prefix
-us option 2-43
Syntax checking only
-y option 2-47
Syntax, compiler 2-1
System data 9-37
System functions 5-6, 9-33
close 5-8, 9-33
creat 5-9
relationship to close 5-8
relationship to read 5-30
relationship to write 5-37
_exit 5-11
lseek 5-19, 9-33
open 5-26, 9-33
relationship to close 5-8
relationship to read 5-30
relationship to write 5-37
read 5-30, 9-33
sbrk 5-31
unlink 5-36, 9-33
write 5-37, 9-33

T

Tagging functions
-Kt option 2-28
tags section 2-16, 2-28, 8-6
tan function 2-47
tanh function 2-47
Temporary files
UNIX
alternate locations 2-4
Termination
normal program 5-11
test_lib script 4-8
test_one script 4-8
Threads
definition 9-11
multiple 9-11
Time optimization
-Ot option 2-37
_ _TIME_ _ preprocessor symbol 3-4
Time stamp
compilation
-Vt option 2-44
Title for listing file
-Flt option 2-20
toascii function 5-33
_tolower function 5-34
tolower function 5-2
Toolkit components 1-1
_toupper function 5-35
toupper function 5-2
Trailer bytes 6-20
Truncating identifiers
-ut option 2-43

Type casting 6-7
Type conversion 6-5
typeof operator 2-54
Types
(see also Data, types)
returned for functions 9-5

U

-U option 2-42
-ui option 2-43
Undefining preprocessor macros
-U option 2-42
Uninitialized data section 2-16
Uninitialized global variables 2-45
Uninitialized static data section
naming
-NZ option 2-32
Union return value 7-5
union type 6-5
UNIX
command line 2-1
compiler syntax 2-1
compiler use 2-1
compiling a program 2-65
environment variables 2-3
file locations 2-3
file name defaults 2-2
invoking the compiler 2-1
libraries provided 2-60
option form, positive and negative 2-6
option, command line
descriptions of 2-6
system functions 2-63
_UNIX preprocessor symbol 3-6
unlink function 2-64, 5-36, 9-33
Unlinking a file name 5-36
unpacked keyword 2-54
unpacked type 6-23
Unreachable (dead) code elimination 10-2
Unsigned char default
-Ku option 2-28
Unsigned integer
conversion to ASCII string 5-18
Unsigned long integer
conversion to ASCII string 5-21
unsigned types 6-4
-upd option 2-43
Upper-case characters
converting to 5-35
-upu option 2-43
-us option 2-43
User-modified routines
for embedded systems 9-31
USR_MRI 2-4, 2-5, 2-24, 2-25
-ut option 2-43
-ut0 option 2-43
-utn option 3-7

V

-v option 2-44, A-20
Value comparison
returning the greater of two values 5-22
returning the lesser of two values 5-25
Variable initialization
-KI option 2-26
Variable names, truncating
-ut option 2-43
Variables
allocation of variables 6-26
global, uninitialized 2-45
initializations 9-40
local 6-27
in the function prologue 7-8
names 7-12
external 11-1
inside asm 9-6
public 11-1
static 11-2
saving initialized variables in ROM 9-40
static 5-3, 6-27
structure members, aligning
-Z options 2-47
vars section
specifying address mode
-ai options 2-17
-Vb option 2-44

-Vd option 2-44
Verbose mode, enabling and disabling
    -V options 2-44
_VERSION preprocessor symbol 3-4
-Vi option 2-44
volatile variables, disabling optimizations
    -Ob option 2-34
vsprintf function 9-11
-Vt option 2-44
-Vw option 2-44

W

-Wa option 2-10, 2-14, 2-44
#warning directive 3-22
    relation to #pragma warn 3-21
Warning messages A-1
    suppression
        -Qw option 2-41
WARNING-xxx (stub routine) called 9-33
    ftell 5-19, 5-26, 5-36
_WARNING_xxx_stub_used symbol
    unresolved 9-33
        lseek 5-19
        open 5-26
        unlink 5-36
_WCHAR_T preprocessor symbol 3-4
Weak externals 2-45
    initialize to zero
        -X0 option 2-46
    no value
        -Xp option 2-46
    uninitialized globals
        -Xc option 2-45
-Wl option 2-45
Word size 6-14
write function 2-64, 5-37, 9-33
Writing a short integer to a stream 5-29
Writing bytes to a file 5-37

X

-x option 2-46, 3-7
-X0 option 2-46
-Xc option 2-45
-Xp option 2-46
XRAY Debugger
    description 1-3
    fully qualified path names
        -Gf option 2-21
    generating debugging information
        -g option 2-22
    line number information
        -Gl option 2-21
    multiple statements on line
        -Gm option 2-22
    preprpcessor macros
        -Gd option 2-21
    restricted information
        -Gr option 2-22
XRAY MasterWorks 1-4
XRAY Source Explorer
    -GS option 2-22
    -Gs option 2-22
XSEG, constant section 2-31

Y

-y option 2-47

Z

-Z2 option 2-47
-Z4 option 2-47
zalloc function 5-6, 5-38, 9-12
-zc option 2-48
-Ze option 2-29
-ze option 2-48
Zero initialization
    local variables
        -KI option 2-26
zerovars section 2-16, 6-27, 8-5
    naming
        -NZ option 2-32
-Zi option 9-8
-Zm option 2-48
-Zn option 2-48

-Zp2 option 2-29, 6-8
-Zp4 option 2-29, 6-8

Introduction 1

Components of the Microtec MCC68K Package

The MCC68K ANSI C Compiler is part of the Microtec software development toolkit for the Motorola 68000 family of microprocessors. Other components include a relocatable cross assembler, a linker, and an object module librarian. The XRAY Debugger is available as an option.

The components of the Microtec 68000 family toolkit are described in more detail in the following sections.

MCC68K Compiler

The MCC68K ANSI C Compiler converts both ANSI C and traditional C source programs into tight, efficient assembly language code for the ASM68K Assembler.

The Microtec C compiler package provides the following features:

- Implementation of the ANSI C and the traditional C programming language
- Ability to compile large C source modules
- Options to specify search paths for standard and nonstandard **#include** files
- Ability to generate optimized code for all members of the 68000 family
- Ability to generate reentrant code
- Ability to produce highly optimized assembly code
- Run-time library that supports all ANSI C functions applicable to embedded systems
- A simple interface to assembly language
- Support for in-line assembly instructions including referencing program variables by their C names
- Ability to generate a listing file containing C code intermixed with assembly code
- Support for source-level debugging of optimized code with the XRAY Debugger

- Ability to locate code and constants in ROM and data in RAM
- Ability to generate line number and symbolic debugging information for use with the Microtec XRAY Debugger
- Generation of warning messages
- Ability to load only those library modules needed

The MCC68K Compiler contains a standard ANSI C preprocessor, which you can run independently. The compiler can produce information necessary for debugging with the XRAY68K Debugger.

CCC68K C++ Compiler

The Microtec C++ compiler is designed to assist in the development of efficient, portable, and easy-to-maintain C++ programs for microprocessor-based systems. It has many features that make it particularly suitable for generating code in an embedded system environment. These features include the ability to produce ROM-able code, position-independent code, and reentrant code. They allow interrupt handlers to be written in C++. The C++ compiler is source code compatible with the Microtec C compiler, except for the differences in standard C and C++ language definitions; thus, all of the standard Microtec features are supported.

ASM68K Assembler

The ASM68K Assembler converts assembly language programs into relocatable object modules conforming to the IEEE-695 standard format. Object modules are suitable for linking with other object modules or with libraries.

The ASM68K Assembler processes macros and conditional assembly statements. The assembler generates a cross-reference table as well as a standard symbol table. Code and data can be placed in multiple named or numbered sections.

After assembly, LNK68K links the object modules, which can then be downloaded and run on any 68000 family microprocessor or the XRAY Debugger (if the modules are in the appropriate format).

LNK68K Linker

The LNK68K Linker combines relocatable object modules into a single absolute object module that can be in the following formats:

- HP 64000 Object Module Format (HP-OMF)
- Microtec extended IEEE-695 format
- Motorola S-record format

Additionally, the linker combines multiple relocatable object modules into a single relocatable module, which subsequently can be relinked with other modules. This feature is called incremental linking.

If one of the input files is a library of object modules, the linker automatically loads any modules from that library that are referenced by the other named object modules. The linker produces a link map that shows the final location of all modules and sections and the final absolute values of all symbols. A cross-reference listing shows which modules refer to each global symbol.

You can run the linker from the command line or in batch mode by using a command file. The linker reports unresolved external symbols, as well as errors that occur at link time, and prints these messages on the link map or on the screen.

LIB68K Object Module Librarian

The LIB68K Librarian creates and maintains program libraries. A program library is a file containing relocatable object modules.

Program libraries let you automatically load frequently used object modules without concern for the specific names and characteristics of the modules. The LIB68K Librarian program lets you format and organize library files that will subsequently be used by the LNK68K Linker.

XRAY68K Debugger (Optional)

The XRAY68K Debugger lets you monitor and control execution of programs at the source and assembly level. You can examine or modify the value of program variables using the same source-level terms, definitions, and structures that were defined in the original source code. To display information, the XRAY68K Debugger employs a window-oriented user interface that divides debugging information into windows.

The debugger gives you complete interactive control of the program by directing an execution environment such as a simulator, in-circuit emulator, or target monitor. The debugger executes your program while allowing you to access variables, procedures, source line addresses, and other program entities.

The powerful XRAY68K Debugger command language allows simple and complex breakpoint setting, single-stepping, and continuous variable monitoring. Input and output can be directed to/from files, buffers, or windows. In addition, a sophisticated macro facility lets you automate repetitive tasks and associate complex command sequences with events such as the execution of a specific statement or the accessing of a specific data location. These features allow you to isolate errors and patch your source code.

XRAY Pro Debug Suite

The XRAY Pro debug suite is an integrated set of tools designed to meet the needs of embedded systems programmers. The XRAY Pro software development environment binds Microtec tools into a system controlled through a mouse-driven, graphical user interface. The integration of Microtec tools provides the following benefits:

- Coverage for each stage of the product development cycle. Once you enter XRAY Pro, you never have to exit to perform a necessary task.
- Common look and feel among all XRAY Pro tools. You only need to learn one interface.
- Communication between tools. Because all tools in the development environment can send messages to each other, each tool has the most current information.

Data Flow

Figure 1-1 shows the components of the Microtec 68000 family software development toolkit and the flow of data among them.

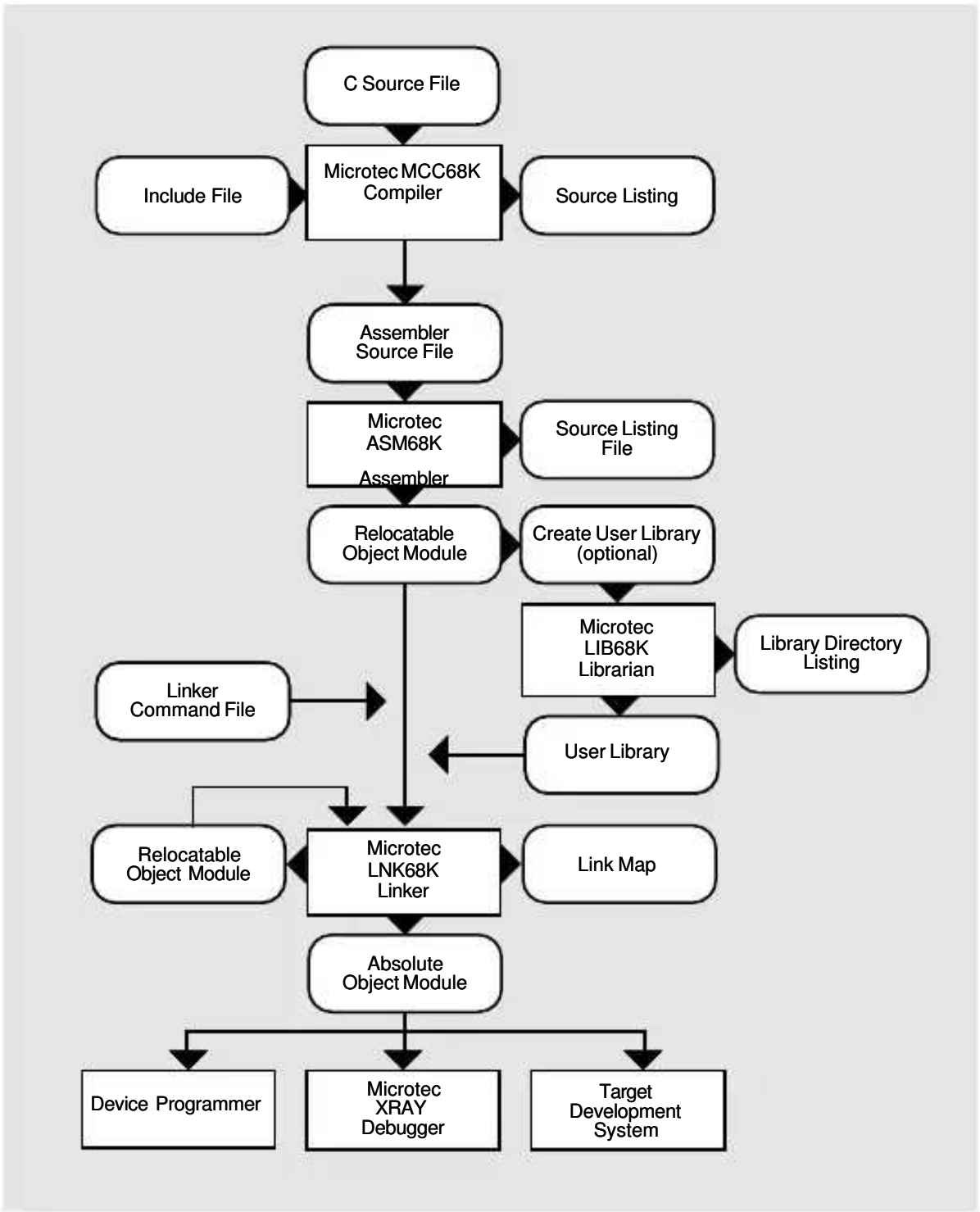

Figure 1-1. Cross Environment Data Flow Through the 68000 Family Toolkit

UNIX/DOS User's Guide 2

This chapter describes how to use the Microtec MCC68K C Compiler on a host computer running a DOS or UNIX operating system.

This chapter includes:

- A description of the syntax to invoke the compiler
- A quick reference table summarizing the command line options
- Descriptions of command line options and defaults
- Example command lines making use of various options

Note

This document uses the term "68020-class" to refer to the 68020, 68EC020, 68030, 68EC030, 68040, 68040V, 68EC040, 68LC040, 68060, 68EC060, and the CPU32/CPU32+ families of microprocessors: 68330, 68331, 68332, 68333, 68340, 68349, and 68360.

Invoking the Compilation Driver

The MCC68K ANSI C Compiler is distributed with a UNIX-style driver program that can conveniently invoke the MCC68K C Compiler, ASM68K Assembler, and LNK68K Linker.

The driver program accepts multiple input files and automatically invokes the appropriate sequence of compiling, assembling, and linking commands to produce an executable file.

This section describes the compiler's invocation command, input and output file name extensions, and default file locations.

Command Line Syntax

The following command invokes the compilation driver:

`mcc68k [-`*option* | *source_filename*`] ...`

Description:

`mcc68k` The name of the compilation driver.

option Any of the command line options described in the section *Command Line Options* in this chapter. You may specify *option*s and *source_filename*s in any order. A dash (**-**) must precede each option. If no options are specified, the driver uses the default settings.

source_filename The name of a file containing one of these:

- C program source file
- 68000 family assembly language file
- Object file
- Object library

File Name Extensions

For each of the input files specified on the command line, the driver determines the file type and invokes either the compiler, assembler, or linker based on the file type. The driver determines the type of each input file based on the file's extension.

Table 2-1 lists the extensions for input files.

Table 2-1. Default Input File Name Extensions

File	UNIX Extension	DOS Extension
C intermediate preprocessed file	.i	.i
C source file (from user)	.c	.c
Assembly language file	.s	.src or .asm
Relocatable object file	.o	.obj
Object file library	.lib	.lib
Linker command file	.cmd	.cmd

The driver uses the following rules to determine how to process each file:

- If a file extension is **.c** or **.i**, it is compiled with the MCC68K Compiler.
- If a file extension is **.s**, **.src**, or **.asm**, it is assembled with the ASM68K Assembler.

- If a file has any other extension (or if it has no extension), the driver assumes it is an object file or a library, and it is given to the LNK68K Linker. The driver will display a warning message if the file does not have a **.o**, **.obj**, or **.lib** extension.

When an output file is created by the compiler, assembler, or linker, it is given a default extension based on its type.

Table 2-2 lists the default extensions for output files.

Table 2-2. Default Output File Name Extensions

File	UNIX Extension	DOS Extension
Preprocessor output file	.i	.i
Assembly language file	.s	.src
Relocatable object file	.o	.obj
Executable file	.x	.abs

Input and Output File Locations

If an input file name does not include a directory specification, the driver looks for the file in the current directory.

If an output file name is not specified or if an output file does not include a directory specification, the file is placed in the current directory.

Environment Variables

The driver and compiler have default directories that they will use to write temporary files and to find toolkit executables, run-time libraries, and standard include files.

To override these default directories, set the environment variables listed in Table 2-3. The environment variables **MRI_68K_BIN**, **MRI_68K_LIB**, and **MRI_68K_INC** override the **USR_MRI** variable. Except for **USR_MRI**, multiple directories can be specified by separating entries with a semicolon(**;**). This establishes the order in which the directories you select will be searched.

For a list of all environment variables that affect the search paths for other Microtec products, refer to *Getting Started (Unix Hosts)* (102046-006).

Table 2-3. Default Environment Variables and Defaults

Environment Variable	Description	UNIX Default	DOS Default
USR_MRI	Directory under which executables, include files, and library directories can be found	/usr/mri	\microtec
MRI_68K_BIN	Directories to search for executable files	**$USR_MRI/bin** or **/usr/mri/bin**	**$USR_MRI\bin** or **\microtec\bin**
MRI_68K_INC	Directories to search for standard include files	**$USR_MRI/include/mcc68k** or **/usr/mri/include/mcc68k**	**$USR_MRI\include\mcc68k** or **\microtec\include\mcc68k**
MRI_68K_LIB	Directories to search for libraries and the default linker command file	**$USR_MRI/lib** or **/usr/mri/lib**	**$USR_MRI\lib** or **\microtec\lib**
MRI_68K_TMP	Directories to use for temporary files	**$TMP** or **/tmp**	Value of **TMP** environment variable or current directory

Note

If you invoke the compiler using an absolute pathname (e.g., **/usr/mri/bin/mcc68k**), then the driver will ignore the **$USR_MRI** environment variable when looking for other executables, such as the linker and assembler. Instead, it will use the path specified when invoking the compiler (in the previous example, **/usr/mri/bin**). However, if the **$MRI_68K_BIN** environment variable is set, that value will override this behavior.

Setting Environment Variables for UNIX

The following examples show how to set the environment variables.

Example:

If the distribution files have been installed under **/usr2/mri**, follow these steps:

1. Include **/usr2/mri/bin** in your current search path.
2. Set the **USR_MRI** environment variable.

   If you are using the C shell (**csh**), enter the command:

   ```
   setenv USR_MRI /usr2/mri
   ```

   If you are using the Bourne shell (**sh**) or Korn shell (**ksh**), enter these commands:

   ```
   USR_MRI=/usr2/mri
   export USR_MRI
   ```

   The above commands set the search path for executable files to **/usr2/mri/bin**, for include files to **/usr2/mri/include/mcc68k**, and for library files to **/usr2/mri/lib**.

Example:

If the executable files are in a directory other than **/usr/mri/bin** or **$USR_MRI/bin** (for example, **/usr3/mri/bin**), follow these steps:

1. Include **/usr3/mri/bin** in your current search path.
2. Set the **MRI_68K_BIN** environment variable.

   If you are using the C shell (**csh**), enter the command:

   ```
   setenv MRI_68K_BIN /usr3/mri/bin
   ```

   If you are using the Bourne shell (**sh**) or Korn shell (**ksh**), enter these commands:

   ```
   MRI_68K_BIN=/usr3/mri/bin
   export MRI_68K_BIN
   ```

Set **MRI_68K_INC**, **MRI_68K_LIB**, and **MRI_68K_TMP** in a similar manner.

Setting Environment Variables for DOS

Environment variables are automatically set during installation using NT registry. The default directory structure is similar to that of UNIX; see Table 2-3 for more details about the locations of NT files.

Command Line Options

Command line options specify the names of output files and turn features on and off.

Options are preceded by a minus sign (**-**).

A single-letter option without arguments may be followed immediately by another option letter. For example, the combination of options **-c** and **-g** can be written as **-cg**.

Some options have a negative form. The negative form disables or turns off the option. You can specify the negative form by entering **n** before the name of the option.

Examples:

`-A`	Set ANSI-compliant mode (ANSI extensions enabled)
`-nA`	Non-ANSI mode (no ANSI extensions enabled)

If conflicting options are specified in the same command line, the rightmost specified option takes effect. For example, if a command line includes two file names specified for the same output file, the second name will be used.

All of the following command line options can be given to the driver program. Some of the options control the behavior of the driver itself. Most of the options, however, do not pertain directly to the driver program and are simply passed on to the compiler.

Command Line Option Summary

Table 2-4 summarizes the positive form of MCC68K command line options.

Table 2-4. UNIX/DOS Command Line Option Summary

Option	Meaning
-A	Sets ANSI-compliant mode. (default: **-A**)
-acc	References items in the **const** section according to the **-Mc** option. (default: **-acc**)
-acd	References items in the **const** section according to the **-Md** option. (default: **-acc**)
-aic	References items in the **vars** section according to the **-Mc** option. (default: **-aid**)
-aid	References items in the **vars** section according to the **-Md** option. (default: **-aid**)
-alc	References items in the compiler-generated **literals** sections according to the **-Mc** option. (default: **-alc**)
-ald	References items in the compiler-generated **literals** sections according to the **-Md** option. (default: **-alc**)
-asc	References items in the **strings** section according to the **-Mc** option. (default: **-asc**)
-asd	References items in the **strings** section according to the **-Md** option. (default: **-asc**)
-C	Saves comments in preprocessor output. (default: **-nC**)
-c	Produces an object file but not an executable file. (default: **-nc**)
-D*name*[=*value*]	Defines the value of a preprocessor macro. (default: no preprocessor macro values supplied)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-d*option_file*	Reads options from the specified file. (default: options supplied by command line)
-E[s]	Invokes the preprocessor only without removing **#line** or **#pragma** directives. Sends output to standard output. (default: **-nE**)
-e*command_file*	Passes the command file to the linker. (default: **-emcc68k.cmd**)
-Fee	Writes diagnostic messages to standard error. (default: **-Fee** for DOS systems) (default: **-nFee** for UNIX systems)
-Feo	Writes diagnostic messages to standard output. (default: **-nFeo** for DOS systems) (default: **-Feo** for UNIX systems)
-Fli	Shows the contents of **#include** files in the listing file. (default: **-nFli**)
-Flp*number*	Sets the page length of the listing file. (default: **-Flp55**)
-Flt*string*	Specifies the title for the listing file. (default: no title string appears on the listing file)
-Fsi	Shows the contents of each **#include** file in the assembler source file. (default: **-nFsi**)
-Fsm	Includes high-level source code as comments in assembly output file. (default: **-nFsm**)
-f	Generates code that uses instructions provided by a floating-point coprocessor. (default: **-nf**)
-Gd	Generates debug information for preprocessor macros. (default: **-Gd**)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-Gf	Generates fully qualified path names for input files. (default: **-nGf**)
-Gl	Generates line number labels. (default: **-nGl**)
-Gm	Generates debugging information for multistatement lines. (default: **-Gm**)
-Gr	Generates restricted debugging information. (default: **-nGr**)
-Gs	Generates data for use by the XRAY Source Explorer. (default: **-nGs**)
-GS	Enables the data browsing facility of the XRAY Source Explorer. (default: **-nGS**)
-g	Generates debugging information. (default: **-ng**)
-H	Saves assembly file. (default: **-nH**)
-h	Generates code for HP 64000 Series Development Systems. (default: **-nh**)
-I*dir*	Specifies the search path for nonstandard **#include** files. (default: see the full description of this option in this chapter for a list of the directories searched)
-I@	Changes the location of the source file's directory on the include search path.
-J*dir*	Specifies the search path for standard **#include** files. (default: see the full description of this option in this chapter for a list of the directories searched)
-Kc	Directs the compiler to use the **CLR** instruction. (default: **-Kc**)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-Kf	Directs the compiler to generate a stack frame for all functions. (default: **-nKf**)
-Kh*reg*[,*reg*] . . .	Reserves registers. (default: no registers reserved)
-KI	Initializes local variables to zero. (default: **-nKI**)
-Km	Generates messenger symbols. (default: **-Km**)
-KP	Forces all structures and unions to be packed. (default: **-nKP**)
-Kq	Makes double variables or constants 32 bits in size. (default: **-nKq)**
-Kr	Uses the return-from-procedure instruction to return from an interrupt function. (default: **-nKr**)
-Kt	Generates code to tag a function's exit and entry points. (default: **-nKt**)
-Ku	Treats plain **char** variables as **unsigned**. (default: **-nKu**)
-Kv	Treats bit fields as **unsigned**. (default: **-nKv**)
-l[*filename*]	Generates a source listing with errors. (default: **-nl**)
-Mca	Directs the compiler to use absolute addressing for all code references. (default: **-Mca**)
-Mcp	Directs the compiler to use PC-relative addressing for all code references. (default: **-Mca**)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-Mda	Directs the compiler to use absolute addressing for all data references. (default: **-Mda**)
-Md*n*	Directs the compiler to use register-relative addressing for all static data references. (default: **-Mda**)
-Mdp	Directs the compiler to use PC-relative addressing for all static data references. (default: **-Mda**)
-Ml	Directs the compiler to use 32-bit displacement for **a***n*- or PC-relative addressing. This option only applies to processors using the 68020-class instruction set. (default: 16-bit displacement)
-NC*name*	Sets the constant variables section name. (default: **-NCconst**)
-NI*name*	Sets the initialized data section name. (default: **-NIvars**)
-NL*name*	Sets the compiler-generated literals section name. (default: **-NLliterals**)
-NM*name*	Sets the module name. (default: module name is source file name stripped of any suffix or prefix)
-NS*name*	Sets the string section name. (default: **-NSstrings**)
-NT*name*	Sets the code section name. (default: **-NTcode**)
-NZ*name*	Sets the static uninitialized static data section name. (default: **-NZzerovars**)
-O	Enables a standard set of optimizations and lists options selected. (default: **-nO**)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-Ob	Performs optimizations that assume that memory will not be changed externally. (default: **-nOb**)
-Oc	Does not pop the stack after each function call. (default: **-Oc**)
-Oe	Generates only one exit (return) from a function. (default: **-nOe**)
-Og	Enables global-flow optimizations. (default: **-nOg**)
-Oi	In-lines function calls within a module. (default: **-nOi**)
-Oj	Enables runtime library function in-lining. (default: **-Oj**)
-Ol	Enables local optimizations. (default: **-Ol**)
-OR	Allocates heavily used variables to registers. (default: **-OR**)
-Or	Enables instruction scheduling optimization. (default: **-nOr**)
-Os	Optimizes in favor of code size rather than execution time. (default: **-nOs**)
-Ot	Optimizes in favor of execution time rather than code size. (default: **-nOt**)
-o*filename*	Names the output file. (default: see full entry for more information)
-P[s]	Executes the preprocessor only, sending output to the **.i** file. (default: **-nP**)
-p*processor*	Produces code for a specified processor. (default: **-p68000**)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-Q*number*	Specifies the maximum number of error messages before quitting. (default: **-Q20**)
-Qe	Suppresses error, warning, and informational messages. (default: no messages suppressed)
-Qi	Suppresses informational messages. (default: **-Qi**)
-Qo	Suppresses listing of options currently active. (default: **-nQo**)
-Qs	Suppresses summary messages. (default: no messages suppressed)
-Qw	Suppresses warning and informational messages. (default: no messages suppressed)
-q	Passes default libraries to linker.
-S	Generates code for the assembler. (default: no assembly language file generated)
-U*name*	Undefines a preprocessor macro. (default: no preprocessor macro is undefined)
-uau	Appends an underscore to all public symbols. (default: -**upu**)
-ui[*char*]	Changes the default insert character for **asm** pseudofunction calls from a back quote (`) to *char*. (default: back quote (`) character used)
-upd	Prepends a dot to symbol names. (default: **-upu**)
-upu	Prepends an underscore to symbol names. (default: **-upu**)
-us	Suppresses the name modifier. (default: **-upu**)
-ut*number*	Truncates the length of program identifiers to *number*. (default: **-ut0**)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-Vb	Displays the banner before compiling. (default: banner not displayed)
-Vd	Displays commands for invoking components. Commands are not executed. (default: commands executed but not displayed)
-Vi	Displays commands for invoking components. Commands are executed. (default: commands executed but not displayed)
-Vt	Displays a time stamp for the various stages of the compilation (UNIX only). (default: time stamps not displayed)
-Vw	Displays the banner and exits. (default: compilation occurs without displaying banner)
-v	Detects and issues warnings for features in the source files that can cause problems. (default: **-nv**)
-Wa,'*option1* [,*option2*,] . . . '	Passes options to the assembler. (default: no options passed to assembler)
-Wl,'*option1* [,*option2*,] . . . '	Passes options to the linker. (default: no options passed to linker)
-Xc	Treats uninitialized global variables as "weak externals." (default: **-Xc**)
-Xp	Allocates space for global variables that have not been explicitly initialized. (default: **-Xc**)
-X0	Allocates space for global variables that have not been explicitly initialized and initializes them to zero. (default: **-Xc**)
-x	Enables Microtec extensions to the C language. (default: **-x**)

(cont.)

Table 2-4. UNIX/DOS Command Line Option Summary (cont.)

Option	Meaning
-y	Checks source syntax but does not compile. (default: **-ny**)
-Za2	Uses even-alignment for structure members larger than **char**. (default: **-Za2** for processors with a default alignment of 2)
-Za4	Uses quad-alignment for structure members larger than **short**. (default: **-Za4** for processors with a default alignment of 4)
-Ze	Passes parameter of size **char** on even stack addresses. (default: **-nZe**)
-Zm*number*	Forces packed structure sizes to be a multiple of *number*. (default: **-Zm1**)
-Zn*number*	Forces unpacked struct **sizeof** to be a multiple of *number*.
-Zo	Passes parameters of size **char** on odd stack addresses. (default: **-nZo**)
-Zp2	Passes parameters of size **short** or smaller as **short**. (default: **-Zp4**)
-Zp4	Passes parameters smaller than **int** as **int**. (default: **-Zp4**)
-zc	Instructs C compiler to accept C++ style comments.
-ze	Generates function tables for Exception Handling. (default: no function tables created)

The following pages describe, in alphabetical order, the command line options that are allowed on the UNIX and DOS operating systems for MCC68K and its driver.

Set ANSI-Compliant Mode

-A (default)
-nA

The **-A** option enables additional features provided by ANSI C. The _ _**STDC**_ _ preprocessor symbol is predefined.

The **-nA** option disables ANSI features. With the **-nA** option, the compiler will accept pre-ANSI programs. For example, if **-nA** is used, **volatile** will not be recognized as a keyword.

The **-nA** option does not guarantee compatibility with earlier versions of Microtec compilers or with other pre-ANSI compilers.

Choose Address Mode for Sections

The compiler and linker allocate code and data to the sections listed in Table 2-5.

Table 2-5. UNIX/DOS Allocation of Code and Data

Contents	Section Name
Code	code
Compiler-generated C++ static constructor/destructor	initfini
Compiler-generated C++ initializers for position-independent virtual tables	pixinit
Compiler-generated literals	literals
Compiler-generated tag data	tags (**-Kt** option only)
Initialized data	vars
Strings	strings
Uninitialized data	zerovars

Items in **code** are referenced according to the address mode chosen with the **-Mc** options (**-Mca** or **-Mcp**).

Items in **zerovars** and **tags** are referenced according to the address mode chosen with the **-Md** options (**-Mda**, **-Mdp**, or **-Md***n*).

You can reference items in the remaining sections (**vars**, **const**, **strings**, **literals**) according to the address mode chosen with the options **-Mc** or **-Md**, depending on

where the sections will be placed at link time. If a section is to be placed in ROM (with **code**), its items should be referenced according to the **-Mc** options. If a section is to be placed in RAM (with **zerovars** and **vars**), its items should be referenced according to the **-Md** option.

-acc (default)
-acd

The **-acc** option references items in the constant variables section according to the addressing option specified for all code references.

The **-acc** option references items in **const** according to the **-Mc** option.

The **-acd** option references items in **const** according to the **-Md** option.

-aic
-aid (default)

The **-aic** option references items in the initialized data section according to the addressing option specified for all code references.

The **-aic** option references items in **vars** according to the **-Mc** option.

The **-aid** option references items in **vars** according to the **-Md** option.

-alc (default)
-ald

The **-alc** option references items in **literals** according to the **-Mc** option.

The **-ald** option references items in **literals** according to the **-Md** option.

-asc (default)

The **-asc** option references items in **strings** according to the **-Mc** option.

-asd

The **-asd** option references items in **strings** according to the **-Md** option.

Save Comments

-C
-nC (default)

The **-C** option displays high-level source comments in the preprocessor output.

The **-nC** option excludes comments from the preprocessor output.

Produce Object File Only (Driver Option)

-c
-nc (default)

The **-c** option produces only an object file (having a **.o** extension for UNIX and an **.obj** explanation for DOS). It tells the driver to produce an object file but does not call the linker to produce an executable file (**.x** extension for UNIX and **.abs** extension for DOS).

The **-nc** option tells the driver to produce an object file and to call the linker to produce an executable file.

Define a Preprocessor Macro Name

-D*name*[=*value*]

The **-D** option defines the value of a preprocessor macro. If you do not specify a string, the preprocessor macro has the value 1 (equivalent to putting **#define** *name* **1** at the top of the source file).

You can define multiple names by preceding each name with **-D**. For example:

```
-Dabc -DABC
```

is equivalent to using the preprocessor directives **#define abc 1** and **#define ABC 1**.

You can also use the **-D** option to specify that structures, unions, or enumerated data types are packed by using the **packed** keyword:

```
-Dstruct="packed struct"
-Dunion="packed union"
-Denum="packed enum"
```

where *struct*, *union*, and *enum* are replaced with the appropriate structure, union, or enumerator names from your program.

Note

The **-U** (undefine macro) option is processed before the **-D** (define macro) option regardless of the order in which they appear on the command line.

Read Options From File (Driver Option)

-d*option_file*

The **-d** option directs the driver to read command line options from the specified file.

Display Preprocessor Output (Driver Option)

-E[s]
-nE (default)

The **-E** option tells the driver to execute the preprocessor only. Preprocessor output is sent to standard output. The **#line** and **#pragma** directives are not removed. The **#include** directives are converted to appropriate **#line** directives so that the original structure of the source files can be determined.

If **-Es** is specified, all the **-E** actions are performed but the echoing of the backslash and newline is disabled.

This option conflicts with **-c**, **-l**, **-P**, and **-S**.

Pass Command File to Linker (Driver Option)

-e*command_file*

The **-e** option directs the driver to pass the specified command file to the linker. This option also prevents the driver from passing the default library and default linker command file (**mcc68k.cmd**) to the linker.

If you wish to use the default library in conjunction with the libraries specified in the linker command file, use the **-q** option with the **-e***command_file* option.

Redirect Diagnostic Messages

-Fee (DOS default)
-nFee (UNIX default)

The **-Fee** option writes diagnostic messages to the standard

error device (**stderr**). If **-nFee** is specified, the diagnostic messages are written to the standard output device. The **-Fee** option conflicts with the **-Feo** option; if both are specified, the rightmost specification takes effect.

-Feo (UNIX default)
-nFeo (DOS default)

The **-Feo** option writes diagnostic messages to the standard output device (**stdout**). If **-nFeo** is specified, the diagnostic messages are written to the standard error device. The **-Feo** option conflicts with the **-Fee** option; if both are specified, the rightmost specification takes effect.

Specify Format of Output Files

-Fli
-nFli (default)

The **-Fli** option puts the contents of each **#include** file in the listing file. You must specify the **-l** option to generate a listing file.

-Flp*number*

The **-Flp** option sets the number of lines per page in the listing file. The default page length is 55 lines. If you specify a page length less than 10, it will be set to 55 lines. You must specify the **-l** option to generate a listing file.

Specifying **-Flp0** omits page breaks and the page header from the listing file.

-Flt*string*

The **-Flt** option specifies a title string that appears at the top of each page of the listing file. You must specify the **-l** option to generate a listing file. Enclose *string* in double quotes if it contains any spaces or punctuation.

-Fsi
-nFsi (default)

The **-Fsi** option expands the contents of each **#include** file in the assembler source file.

-Fsm
-nFsm (default)

The **-Fsm** option includes high-level source code as comments in the assembler source file. You must specify the **-S** or **-H** option in order to generate an assembly file.

Generate Floating-Point Processor Instructions

-f
-nf (default)

The **-f** option generates code that uses the instructions provided by the 68881/68882 floating-point coprocessor. Note that the 68040 processor has its own coprocessor and does not require the use of the **-f** option for floating-point code.

The **-f** option generates the messenger symbol **____FPU**, which is resolved in the run-time library. See the section *Messenger Symbol* in Chapter 7, *Embedded Environments*, of the *Reference Manual* for further information.

The **-nf** option implements all floating-point operations by making calls to the run-time library.

Produce Debugging Information

-Gd (default)
-nGd

The **-Gd** option generates debug information for preprocessor macros.

Use the **-Gd** option in conjunction with the **-g** option.

-Gf
-nGf (default)

The **-Gf** option generates fully qualified file names for input files, in addition to line number and symbol information. A fully qualified file name has a full path name, even if the full path was not given on the command line. The fully qualified file name lets the XRAY Debugger find the source file (**.c**, **.cc**, or **.cxx** extension) even if the XRAY Debugger is not invoked from the directory in which the source file resides.

Use the **-Gf** option in conjunction with the **-g** option.

-Gl
-nGl (default)

The **-Gl** option generates line number labels of the form **LL***nn*, where *nn* is the line number. These line number labels will not be recognized by the XRAY Debugger but can be used in other debugging environments.

The **-Gl** and the **-g** option are mutually exclusive. If both **-Gm** and **-Gl** are specified, **-Gm** is ignored.

-Gm (default)
-nGm

The **-Gm** option generates debugging information for stepping through individual statements on lines containing more than one statement.

This option requires the Microtec XRAY Debugger version 2.2 or later. Earlier versions of the XRAY Debugger may require multiple step commands to get past a line of source code containing more than one statement.

Use the **-Gm** option in conjunction with the **-g** option. If both **-Gm** and **-Gl** are specified, **-Gm** is ignored.

-Gr
-nGr (default)

The **-Gr** option generates "restricted" debugging information. For each function, information concerning its frame type, push mask, and start/end addresses are generated. No debugging information is generated for line numbers or symbols.

Use this option when compiling modules that have already been debugged. The **-Gr** option lets the XRAY Debugger economically provide call tracing information for the functions in those modules. This functionality is present in the XRAY Debugger version 2.2 and subsequent versions.

The **-Gr** option and the **-g** option are mutually exclusive.

-Gs
-nGs (default)

The **-Gs** option generates information for use by the XRAY Source Explorer.

Use the **-Gs** option in conjunction with the **-g** option.

-GS
-nGS (default)

The **-GS** option enables the data browsing facility in the XRAY Source Explorer.

The **-GS** option implies the **-Gs** option.

-g
-ng (default)

The **-g** option generates line number, variable, and symbol information. The **_DEBUG** preprocessor symbol is predefined.

This information is in IEEE-695 format, which the XRAY68K Debugger can read. With **-g**, you must use the

Microtec assembler and linker to pass complete debugging information to the XRAY Debugger.

Save Assembly File (Driver Option)

-H
-nH (default)

The **-H** option instructs the compiler not to remove the generated assembly file.

The name of the assembly file is based on the source file name followed by an extension: **.s** (UNIX), **.asm** (DOS), or **.src** (DOS).

The -**o***filename* option does not affect the name of the assembly file.

The **-nH** option removes the generated assembly file.

Support HP 64000

-h
-nh (default)

The **-h** option generates code which includes two special labels, **R***label* and **E***label*, that HP 64000 emulators use to identify each function's exit and end points for debugging and timing purposes. **R***label* represents the function's return point, and **E***label* represents the function's end address. The *label* portion of the symbol name is the name of the function.

Add Search Path for Nonstandard Include Files

-I*dir*

The **-I** option specifies a search path to be scanned to locate user-supplied **#include** files. If an include file name in your source file is enclosed in double quotes (""), the compiler will search for the include file in the following directories in order:

1. The directory containing the source file that has the **#include** directive.
2. The directory containing the top-level source file.
3. The directories specified by the **-I** option.

   The compiler will search multiple directories if you precede each directory name with **-I**. For example, if you enter **-I***dir1* **-I***dir2* **-I***dir3*, the compiler

searches for **#include** files in the directory containing the source file, the *dir1* directory, the *dir2* directory, and the *dir3* directory, in that order.

4. The directories specified with the **-J** option.
5. The directories specified by the environment variable **MRI_68K_INC**.
6. If **MRI_68K_INC** is not defined, the directory **$USR_MRI/include/mcc68k** on UNIX or the directory specified by the **USR_MRI** environment variable on DOS.
7. The standard include file directory if neither **MRI_68K_INC** nor **USR_MRI** is defined.
8. The directory from which the compiler was invoked.

See the **-J** option for information on include file names enclosed in angle brackets (**< >**), and see the **-I@** option to alter the search order for the include files.

Change Search Path Order for Include Files

-I@
-nI@(default)

The **I@** option overrides the default search order for user-supplied include files.

The **-I@** option indicates that the **#include** files should be searched in the directory containing the current source file and then the top-level source file. While these directories are placed implicitly at the beginning of the search path, this option enables the user to move these directories from the front of the search path to any other position.

The command line:

```
mcc68K -Ia -I@ -Ib c\t.c
```

causes the directory search sequence to be:

1. directory `a`
2. the directory `c` containing the source file
3. directory `b`.

Add Search Path for Standard Include Files

-J*dir*

The **-J** option specifies a search path to be scanned to locate standard **#include** files. When an include file name in your source file is enclosed in angle brackets (**< >**), the compiler will search for the file in the following directories in order:

1. The directories specified by the **-J** option.

   The compiler will search multiple directories if you precede each directory name with **-J**. For example, if you enter **-J***dir1* **-J***dir2* **-J***dir3*, the compiler searches for standard **#include** files in the *dir1* directory, the *dir2* directory, and the *dir3* directory.

2. The directories specified by the environment variable **MRI_68K_INC**.

3. If **MRI_68K_INC** is not defined, the directory **$USR_MRI/include/mcc68k** on UNIX or the directory specified by the **USR_MRI** environment variable on DOS.

4. The standard include file directory if neither **MRI_68K_INC** nor **USR_MRI** is defined.

See the **-I** option for information on include file names enclosed in quotes ("*name*").

Produce Minor Code Generation Variations

-Kc (default)
-nKc

The **-Kc** option tells the compiler to use the **CLR** instruction for assigning zero to register and memory variables.

The **-nKc** option tells the compiler not to use the **CLR** instruction for assigning zero to nonregister variables. On some 68000 family processors, the **CLR** instruction reads memory before writing a zero to the location. This behavior can cause problems for memory-mapped I/O ports.

-Kf
-nKf (default)

The **-Kf** option controls function frame usage.

The **-Kf** option forces frames for all functions. A framed function has a **LINK/UNLK** pair of instructions in the func-

tion prologue and epilogue. In a framed function, variables and parameters are accessed on the stack with the FP-relative addressing mode. The stack pointer is not, however, available for your use; user programs should never modify the stack pointer since this can cause unpredictable run-time errors.

With the **-nKf** option, the compiler will generate a frame for a function only if it is needed.

-Kh*reg*[,*reg*] . . .

The **-Kh***reg* option directs the compiler to reserve the registers specified. The compiler will not generate any code that uses a reserved register.

The value for *reg* can be up to three **A***n* registers (**A2**, **A3**, **A4**, **A5**, **A6**) and up to three **D***n* registers (**D2**, **D3**, **D4**, **D5**, **D6**).

If you enter **-Kh***reg* more than once on the command line, the last occurrence of **-Kh***reg* overrides previous occurrences.

For example, if you enter:

```
-Kha2 -Kha3
```

the `-Kha2` will be discarded and the compiler will not generate any code that uses the `a3` register.

The compiler generates a warning if the reserved register is required to adhere to function calling conventions.

-KI
-nKI (default)

The **-KI** option initializes all local variables to zero. Local variables specifically initialized in your code will subsequently be reinitialized with the specified value. This option does not affect parameters.

The **-KI** option is designed to aid debugging; use of this option can significantly decrease run-time performance for functions containing a large number of local variables or a large, complex local variable.

With the **-nKI** option, no additional code is generated to initialize local variables (user-specified initializations are performed as usual).

-Km (default)
-nKm

The **-Km** option generates messenger symbols.

With the **-nKm** option, messenger symbols are not generated. See the section *Messenger Symbols* in Chapter 7, *Embedded Environments*, for further information.

-KP
-nKP (default)

The **-KP** option forces all structures and unions to be packed.

The **-nKP** option does not pack structures and unions unless explicitly declared packed using the **packed** keyword.

Packed structures are an extension to ANSI C and require Microtec extensions to be enabled using the **-x** option.

-Kq
-nKq (default)

The **-Kq** option makes all double variables or constants 32 bits in size (i.e., the same size as a standard float variable). Since **-Kq** treats long double variables and double, all variables that are float, double, and long double have a size of 4 bytes.

If you wish to use single-precision math, you must rebuild your libraries with the **-Kq** option enabled. All of the new libraries must then be linked to all programs built using **-Kq**.

Note

If the **-Kq** option is used, the pre-built libraries provided in the distribution should not be used, as they treat doubles as 64-bit quantities. The libraries should be rebuilt from the included sources with the **-Kq** option enabled.

-Kr
-nKr (default)

The **-Kr** option specifies that the return-from-procedure (rather than the return-from-interrupt) instruction be used to return from a function declared as an **interrupt** procedure as follows:

- The **-Kr** option uses the **RTS** instruction
- The **-nKr** option uses the **RTE** instruction

-KT
-nKT (default)

The **-KT** option detects misalignments. The preprocessor symbol **_HW_DEMANDS_ALIGNMENT** is predefined when **-KT** is enabled.

The **-KT** option diagnoses potential alignment problems and reports them as warnings.

With the **-nKT** option, the compiler does not check for potential run-time misalignment problems.

-Kt
-nKt (default)

The **-Kt** option generates code to tag each function's entry and exit points. These tags may be used by real-time analysis tools to monitor the execution of the program.

For each function, the compiler will allocate two words of data in the **tags** section. The tag data will be labeled **_r**_*module_func*, where *module* is the name of the module and *func* is the name of the function. For example:

```
                SECTION     tags,,D
                XDEF        _r_myprog_main
_r_myprog_main: DCB.B   4,0
```

The compiler will generate the following code in the prologue (function entry):

```
        move.w      #10,_r_myprog_main
```

The compiler will generate the following code in the epilogue (function exit):

```
        move.w      #100,_r_myprog_main+2
```

The values `10` and `100` are arbitrary. The analysis tool will detect and record writes to the tag data. The values cannot be redefined.

-Ku
-nKu (default)

The **-Ku** option tells the compiler to consider **char** variables declared without the **signed** or **unsigned** keyword as **unsigned**. The preprocessor symbol **_CHAR_UNSIGNED** is predefined when the **-Ku** option is enabled.

With the **-nKu** option, **char** variables without an explicit **signed** or **unsigned** keyword are treated as **signed**, and the preprocessor symbol **_CHAR_SIGNED** is predefined.

-Kv
-nKv (default)

The **-Kv** option tells the compiler to consider bit fields declared without the **signed** or **unsigned** keyword as **unsigned**.

With the **-nKv** option, bit fields without an explicit **signed** or **unsigned** keyword are treated as **signed**.

-Ze
-nZe (default)

The **-Ze** option pushes all parameters of size **char** on to the stack in even memory address values. The **-Ze** option must be used in conjunction with the **-Zp2** option.

-Zo (default)
-nZo

The **-Zo** option pushes all parameters of size **char** on to the stack in odd memory address values. The **-Zo** option must be used in conjunction with the **-Zp2** option.

If the **-Zp2** option is used without the **-Zo** or **-Ze** options, odd addresses will be used for parameters of size **char**.

-Zp2

The **-Zp2** option pushes all parameters of size **short** or smaller on to the stack as 16-bit quantities. If the parameter is smaller than **short**, it is extended to **short**.

The **-nZp2** option is equivalent to **-Zp4**, which extends parameters smaller than **int** to **int**.

If no function prototype is used to indicate the type of the parameters, they are extended to **int** by default.

-Zp4 (default)

The **-Zp4** option extends to **int** all parameters smaller than **int** before pushing them on the stack.

The **-nZp4** option is equivalent to **-Zp2**, which extends parameters of size **short** or smaller to **short**.

If no function prototype is used to indicate the type of the parameters, they are extended to **int** by default.

Generate Listing File

-l[*filename*]
-nl (default)

The **-l** option writes a listing file containing high-level source code and any diagnostic messages to the standard output device or the specified file.

The **-nl** option disables the creation of a listing file.

If the **-l** option is specified with **-P** or **-E**, **-l** is ignored.

Generate Position-Independent Code and Data

-Mca (default)

The **-Mca** option directs the compiler to use absolute addressing for all code references.

-Mcp

The **-Mcp** option directs the compiler to use PC-relative addressing for all code references. This results in code that is position-independent. The preprocessor symbol **_PIC** will be predefined.

-Mda (default)

The **-Mda** option directs the compiler to use absolute addressing for all data references. If you use this option, link with one of the absolute libraries.

-Md*n*

The **-Md***n* option directs the compiler to use register-relative addressing for all references to static data. This results in data that is position independent. The preprocessor symbols **_PID** and **_PID_REG** will be predefined. **_PID_REG** contains the register number *n*.

The **-Md***n* option directs the compiler to use **a***n*-relative addressing. The register number *n* must be in the range 2 to 6. If you use the **a5**-relative addressing option, link with one of the **a5**-relative (**a5**) libraries. For data generated relative to other valid registers, you must build a corresponding library.

See *Chapter 9, Embedded Environments*, for information on modifying the default linker command file and for more information about register-relative global data references.

-Mdp

The **-Mdp** option directs the compiler to use PC-relative addressing for all references to static data. This results in data that is position independent. The preprocessor symbol **_PID** will be predefined.

If you use this option, link with one of the PC-relative (**pc**) libraries.

For more information, refer to the section *Libraries* in this chapter.

-Ml

The **-Ml** option directs the compiler to use 32-bit displacements for **a***n*- or PC-relative addressing. This option only applies to processors supporting a 68020-class instruction set (see Table 2-7 under the **-p** option).

By default, the compiler uses 16-bit displacements for **A***n*- or PC-relative addressing.

Note

The options **-Mca** and **-Mcp** are mutually exclusive.

The options **-Mda**, **-Mdp**, and **-Md***n* are mutually exclusive.

The rightmost option takes effect if conflicting options are specified on the same command line.

Name the Sections

The compiler allocates code and data to sections. Default section names are overridden by these options.

-NC*name*

The **-NC** option sets the constant variables section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (**?**), period (**.**), or underscore (_).

No error message is generated if you give the compiler an illegal name.
(default: **const**)

-NI*name*

The **-NI** option sets the initialized data section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (**?**), period (**.**), or underscore (_).

No error message is generated if you give the compiler an illegal name.
(default: **vars**)

-NL*name* The **-NL** option sets the compiler-generated literals section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (**?**), period (**.**), or underscore (_).

No error message is generated if you give the compiler an illegal name.
(default: **literals**)

-NM*name* The **-NM** option sets the module name.

name can be any sequence of characters.

No error message is generated if you give the compiler an illegal name.
(default: source file name stripped of any suffix or prefix)

-NS*name* The **-NS** option sets the string section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (**?**), period (**.**), or underscore (_).

No error message is generated if you give the compiler an illegal name.
(default: **strings**)

-NT*name* The **-NT** option sets the code section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (**?**), period (**.**), or underscore (_).

No error message is generated if you give the compiler an illegal name.
(default: **code**)

-NZ*name* The **-NZ** option sets the static uninitialized data section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (**?**), period (**.**), or underscore (_).

No error message is generated if you give the compiler an illegal name.
(default: **zerovars**)

Note

If you compile your C source file with the **-Xc** option, the linker will allocate all uninitialized global data in the **zerovars** section. This section can be renamed at link time using the LNK68K **ALIAS** command. See the *Assembler/Linker/Librarian User's Guide and Reference for the 68000 Family* for more information.

If you compile your C source file with the **-Xp** option, you can rename the **zerovars** section at compile time.

Optimize Code

Note

To turn on all optimizations, use the following options:

```
-O -Ob -Oe -Os
```

or

```
-O -Ob -Oe -Ot
```

To turn off all optimizations, use the following options:

```
-nOc -nOl -nOR
```

The rightmost option takes effect if conflicting options are specified on the same command line.

-O
-nO (default)

The **-O** option turns on the following optimizations:

- -Og Global-flow optimizer
- -Oi In-lining optimization
- -Or Instruction scheduling optimization

This option does not turn on all optimizations.

The **-nO** option turns off the previously defined optimizations as well as the **-Oc** and **-Ol** optimizations.

-Ob
-nOb (default)

If the global optimizer (**-Og**) is used and the **-Ob** option is specified, the compiler performs optimizations on global variables. No optimizations will ever be performed on variables declared as **volatile**.

The optimizations performed assume that global variables can only be changed by an unaliased reference to that variable. A variable is considered to be aliased if the address is assigned to a pointer variable.

For a loop invariant expression containing a global variable, the optimizer will move the expression out of the loop even if the loop contains a pointer dereference on the left-hand side of an assignment statement.

With the **-nOb** option, the compiler may still perform optimizations on a non-**volatile** variable as long as there is no pointer dereference on the left-hand side of the assignment statement within the loop in question. Note that **-nOb** does not have the same effect as declaring a variable to be **volatile**.

Warning

The **-Ob** option should not be used on programs that use global variables as operating system memory locations or I/O device registers unless they have been declared **volatile**.

If both the **-Ob** and **-nOg** options are specified, **-nOg** is ignored.

-Oc (default)
-nOc

The **-Oc** option eliminates unnecessary stack processing.

The **-Oc** option does not pop the stack after each function call. The stack will be allowed to grow and will be popped after several function calls. This optimization is known as combining stack pops.

The **-nOc** option generates code to pop the stack after every function call.

If both the **-Oc** and **-nOg** options are specified, **-nOg** is ignored.

-Oe
-nOe (default)

The **-Oe** option generates only one exit (return) from a function. By default, the compiler may generate more than one return instruction if that is more efficient.

-Og
-nOg (default)

The **-Og** option invokes the global-flow optimizer. This option cannot be used in conjunction with the **-nOl** option. If both the **-Og** and **-nOl** options are specified, **-nOl** is ignored.

If both the **-Ob** and **-nOg** options are specified, **-nOg** is ignored.

If both the **-Oc** and **-nOg** options are specified, **-nOg** is ignored.

If both the **-Og** and **-OR** options are specified, **-OR** is ignored.

-Oi
-nOi (default)

The **-Oi** option enables the in-lining optimization; it must be used in conjunction with the **-Og** option.

In-lining is a code generation technique that moves the code of a called function into the caller. This technique eliminates the procedure call overhead (branching and stack allocation) and allows for the possibility of improved code optimization such as constant folding and loop optimizations.

The **-Oi** option tells the compiler to prescan a module looking for functions (usually small functions with only simple local variables) where in-lining can enhance performance without costing too much code space. When the module is compiled, the code for these functions will be expanded in-line.

If an in-lined function invokes another function, that function may also be in-lined.

See the section *Function In-Lining* in Chapter 10, *Optimizations*, for more information.

-Oj (default)
-nOj

The **-Oj** option inlines run-time library functions when Microtec extensions are enabled.

The **-nOj** option disables the in-lining of run-time library functions without disabling other Microtec extensions.

Use the **-Oj** option in conjunction with the **-Og** and **-x** options.

-Ol (default)
-nOl

The **-Ol** option performs local optimizations.

The **-nOl** option disables code hoisting and cross jump optimizations.

This option cannot be used in conjunction with the **-Og** option. If both the **-Og** and **-nOl** options are specified, **-Og** is ignored.

The **-nOl** option implies **-nOc**.
The **-nOl** option implies **-Kf**.

-OR (default)
-nOR

The **-OR** option allocates heavily used variables to registers.

If the global-flow optimizer is used (**-Og**), local variables will be allocated to registers using a register coloring algorithm. If the global-flow optimizer is not used (the default **-nOg**), only heavily used local variables will be allocated to registers (**-OR**).

See Table 2-6 for an explanation of what happens when both **-Og** and **-OR** are specified.

Table 2-6. Interaction of -Og and -OR Options

Option	-nOg (Default)	-Og
-OR (default)	Allocates heavily used local variables to registers.	Local variables will be allocated to registers using a register-coloring algorithm.
-nOR	Disables the allocation of heavily used local variables to registers.	Local variables will be allocated to registers using a register-coloring algorithm.

The **-nOR** option may be used to disable the allocation of heavily used variables to registers.

-Or
-nOr (default)

The **-Or** option enables instruction scheduling optimization. Instruction scheduling rearranges the instruction sequence for improved throughput in the instruction pipeline.

If you use the **-Or** option, the assembly output can be difficult to understand, and some debugging operations can yield confusing results.

The **-nOr** option disables instruction scheduling.

-Os

The **-Os** option optimizes in favor of code size rather than execution time. Using the **-Os** option disables all function inlining.

Use the **-Os** option in conjunction with the **-Og** option.

-Ot

The **-Ot** option optimizes in favor of execution time rather than code size.

Use the **-Ot** option in conjunction with the **-Og** option. If neither the **-Os** nor **-Ot** options are used in conjunction with the **-Og** option, the compiler generates code that is optimized based on compromise between size and speed.

Name the Output File (Driver Option)

-o*filename*

The **-o** option names the output file with the specified name instead of its default name. The output file can be one of these:

Preprocessed file	If used with the **-E** or **-P** options
Assembly source file	If used with the **-S** option
Object file	If used with the **-c** option
Executable file	Assumed by default

By default, the output file name is the source file name stripped of any suffix or prefix.

See the section *Command Line Examples* in this chapter for examples of default output file names and their appropriate extensions.

Send Preprocessor Output to File (Driver Option)

-P[s]
-nP (default)

The **-P** option tells the driver to execute the preprocessor only. Preprocessor output is sent to a file having the same name as the source file but with the suffix **.i**. This option is useful for debugging complicated macro definitions and deeply nested conditional directives.

This option conflicts with **-c**, **-E**, and **-l**.

If **-Ps** is specified, all the **-P** actions are performed, but the

echoing of the backslash and newline is disabled.
With the **-nP** option, compilation continues after preprocessing; the preprocessed source file is not saved.

Refer to Chapter 3, *The Preprocessor*, for more information about the **-P** option and preprocessor directives.

Produce Code for Specified Processor

-p*processor*

The **-p***processor* option produces code for the specified processor.

A predefined preprocessor symbol is defined for each *processor* value.

The possible values for *processor* are **68000** (default), **68008**, **68010**, **68020**, **68030**, **68040**, **68060**, **68302**, **68330**, **68331**, **68332**, **68333**, **68340**, **68349**, **68360**, **68ec000**, **68ec020**, **68ec030**, **68ec040**, **68hc000**, **68ec060**, **68hc001**, **cpu32**, and **cpu32p**.

A predefined preprocessor symbol is defined for each *processor* value.

See section *Predefined Symbols* in Chapter 3, *The Preprocessor*, for a list of symbols.

The **-p***processor* option affects the alignment of global data, local data, and structures. Modules that share structures should be compiled for the same processor unless they are packed.

Some *processor* values actually use the instruction set of a similar processor. The resulting code does not contain any instructions unique to the *processor* specified or instructions not supported by the similar processor. Table 2-7 shows *processor* values, the default alignment, the processor instruction set used, and the library that should be used with the specified *processor*.

Table 2-7. Processor Identification

Processor Value	Default Alignment	Instruction Set Used	Library To Be Used
68000	2	68000	68000
68008	2	68000	68000
68302	2	68000	68000
68EC000	2	68000	68000
68HC000	2	68000	68000
68HC001	2	68000	68000

(cont.)

Table 2-7. Processor Identification (cont.)

Processor Value	Default Alignment	Instruction Set Used	Library To Be Used
68010	2	68010	68000
68020	4	68020	68020
68EC020	4	68020	68020
68030	4	68030	68020
68EC030	4	68EC030	68020
68040	4	68040	68040
68EC040	4	68EC040	68020
68060	4	68060	68040
68EC060	4	68EC060	68020
68330	2	CPU32	CPU32
68331	2	CPU32	CPU32
68332	2	CPU32	CPU32
68333	2	CPU32	CPU32
68340	2	CPU32	CPU32
68349	4	CPU32+	68020
68360	4	CPU32+	68020
CPU32	2	CPU32	CPU32
CPU32P	4	CPU32+	68020

The default alignment also determines whether the **-Za2** (default alignment 2) or **-Za4** (default alignment 4) option is in effect (see section *Modify Alignment* in this chapter for more information).

Refer to section *Alignment and Packing* in Chapter 6, *Internal Data Representation*, for more information on alignment.

If you do not use the default linker command file (**-e** option), you must include the appropriate library for your processor in your linker command file.

> **Note**
>
> This document uses the term "68020-class" to refer to the 68020, 68EC020, 68030, 68EC030, 68040, 68040V, 68EC040, 68LC040, 68060, 68EC060 and the CPU32/CPU32+ families of microprocessors: 68330, 68331, 68332, 68333, 68340, 68349, and 68360.

Suppress Diagnostic Messages

Diagnostic messages have different severity levels: error, warning, or informational.

-Q*number*	The **-Q** option stops compilation if more than *number* errors occur. By default, *number* is 20.
-Qe	The **-Qe** option suppresses error, warning, and informational messages.
-Qi (default)	The **-Qi** option suppresses informational messages.
-Qo -nQo (default)	The **-Qo** option suppresses the listing of options that are currently active (normally produced with **-l** option). The **-nQo** option lists the options that are active on the output listing produced by the **-l** option.
-Qs	The **-Qs** option suppresses the summary of diagnostic messages.
-Qw	The **-Qw** option suppresses warning and informational messages.
-nQ	The **-nQ** option does not suppress any messages.

Pass Default Library to Linker (Driver Option)

-q
-nq (default with **-e***command_file*)

The **-q** option instructs the driver to pass the default library to the linker in addition to the libraries specified in the **-e***command_file* option.

If **-nq** is specified, the compiler will not pass the default libraries to the linker when a linker command file is specified with the **-e***command_file* option.

Produce Assembler Source File (Driver Option)

-S

The **-S** option instructs the driver to produce an assembly language file (**.s** extension for UNIX and **.src** for DOS) which can be passed to the assembler to produce an object file (**.o** extension for UNIX and **.obj** for DOS).

This option conflicts with **-c**, **-E**, **-P**, and **-y**.

Undefine a Preprocessor Macro Name

-U*name*

The **-U** option undefines the defined preprocessor macro specified by *name* (equivalent to putting **#undef** *name* at the beginning of the source file). For example:

```
-UNAME1 -Uname2 -UNAME3
```

undefines the preprocessor macros `NAME1`, `name2`, and `NAME3`.

> **Note**
>
> The **-U** (undefine macro) option is processed before the **-D** (define macro) option regardless of the order in which they appear on the command line.

Modify Naming Conventions

-uau

The **-uau** option appends an underscore to all public symbols.

By default, the compiler prepends an underscore to all public symbols.

-ui[*char*]

The **-ui** option changes the default insert character for **asm** pseudofunction calls from a back quote (`) to *char*. If *char* is null, there is no insert character defined.

See the section *Variable Names Inside asm* in Chapter 9, *Embedded Environments*, for more information on **asm** insertions.

-upd

The **-upd** option (which can also be written as **-up.**) prepends a dot (**.**) to all global symbol names.

-upu (default)

The **-upu** option (which can also be written as **-up_**) directs the compiler to prepend an underscore (_) to all global symbol names.

-us

The **-us** option directs the compiler to suppress the name modifier (dot or underscore).

-ut0 (default)
-ut*number*

The **-ut** option truncates program identifiers to a length of *number*. Language keywords are not truncated.

With the **-ut0** option, program identifiers are not truncated.

Note

The options **-upd**, **-upu**, and **-us** are mutually exclusive.

The compiler's default naming convention is not to prepend an underscore (_) to symbol names. The MCC68K run-time libraries are built using the default naming convention. Therefore, if a module containing a reference to an item in the library is not compiled with the default naming convention, that item will be unresolved at link time.

The compiler's default naming convention is to prepend an underscore to symbol names. The Microtec C run-time libraries and Sun Microsystems' native libraries are built using the default naming convention. Therefore, if a module containing a

reference to an item in the library is not compiled with the default naming convention, that item will be unresolved at link time.

Control Verbose Output Information (Driver Option)

-Vb — The **-Vb** option displays the copyright notice and version number and continues compilation.

-Vd — The **-Vd** option displays the commands that would be used to invoke each component of the toolkit without executing the commands. By redirecting your output to a file, you can use the output to create a script for future compilations.

-Vi — The **-Vi** option displays the commands used to invoke each component of the toolkit as they are executed.

-Vt (UNIX only) — The **-Vt** option displays a time stamp for the various stages of the compilation.

-Vw — The **-Vw** option displays the copyright notice and version number and exits.

-nV (default) — The **-nV** option disables the display of any verbose messages.

Perform Extra Checking

-v
-nv (default) — The **-v** option tells the compiler to issue warnings for certain features in the source file that are likely to be problems. For example, the **-v** option issues warnings for the following among others:

- Functions without prototypes
- Static functions declared but not defined
- **long double** variables treated as **double**

Pass Options to Tools (Driver Option)

-Wa,'*option1*[,*option2*,] . . . '
The **-Wa** option passes the specified options directly to the assembler. This option only applies if the driver is invoking the assembler (i.e., **-E**, **-P**, or **-S** options are not specified).

When passing a simple single command with the **-Wa** option, enclosing quotes are not necessary, as shown in the following example:

```
mcc68k -Wa,-l>main.l main.c
```

Complex embedded commands passed by **-Wa** often require both single (') and double (") quotes as shown in the following example:

```
mcc68k -Vi y.c -Wa,'-f "intfile, case"'
```

In the previous example, both single and double quotes were necessary because the `-f` command line option uses the assembler command line flags `intfile` and `case`.

See the *Assembler/Linker/Librarian User's Guide and Reference for the 68000 Family* for information about assembler command line options.

-Wl,'*option1*[,*option2*,] . . . '

The **-Wl** option passes the specified options directly to the linker. This option only applies if the driver is invoking the linker (i.e., **-c**, **-E**, **-P**, or **-S** options are not specified).

When passing a simple single command with the **-Wl** option, enclosing quotes are not necessary, as shown in the following example:

```
mcc68k -Wl,-m>main.map main.c
```

See the *Assembler/Linker/Librarian User's Guide and Reference for the 68000 Family* for information about linker command line options.

Initialize Uninitialized Global Data

-Xc (default)

The **-Xc** option treats uninitialized global variables as "weak externals" or "C common." A "weak external" is a global-level variable that is not initialized on its declaration. The compiler will not allocate any space for these variables. The linker will resolve each such variable to its initialized declaration, if there is one. For example:

```
int i;
```

If this variable is not externally defined in another module using **XDEF**, the linker will then allocate the variable in the **zerovars** section. This section may be renamed at link time using the LNK68K **ALIAS** command. With the **-Xc** option, weak externals will be output as **XCOM**s.

-Xp

The **-Xp** option allocates space for global variables that have not been explicitly initialized but assigns them no value.

With the **-Xp** option, weak externals will be output as **XDEF**s with the storage defined but not initialized.

The compiler places these variables in the **zerovars** section; this section may be renamed on a per module basis with the **-NZ***name* option. The **zerovars** section is set to zero at program start-up time.

-X0

The **-X0** option initializes to zero all global variables that have not been explicitly initialized when they are allocated.

With the **-X0** option, weak externals will be output as **XDEF**s and initialized to zero. They will be placed in the **vars** section. This section may be renamed on a per module basis with the **-NI***name* option.

Enable Microtec Extensions

-x (default)
-nx

The **-x** option enables the following Microtec extensions to the C language:

- The predefined Microtec preprocessor symbol **_MRI_EXTENSIONS**
- Acceptance of **\X** for hexadecimal sequences
- Additional bit field types **char**, **short**, and **long**
- In-line assembly with **asm**
- In-lining of several library functions
- **interrupt** procedures (**interrupt** keyword)
- **packed** structures (**packed** and **unpacked** keywords) and **enum**s
- Preprocessor directives **#warning** and **#inform**
- **typeof** operator (**typeof** keyword)

The following library functions will be expanded in-line rather than called:

strcpy memcpy strlen

If 68881 code is being generated (**-f**), the following function calls will be replaced by in-line 68881 code:

acos	cosh	sin
asin	exp	sinh
atan	fabs	sqrt
atanh	log	tan
cos	log10	tanh

The **-nx** option disables all Microtec extensions and enables stricter checking as required by ANSI.

Check Program Syntax (Driver Option)

-y
-ny (default)

The **-y** option checks the program syntax without generating code.

The **-ny** option checks the syntax and generates code.

Modify Alignment

-Za2 (default for processors with a default alignment of 2)
-Za4 (default for processors with a default alignment of 4)

The default alignment for structure members depends on the processor selected with the **-p** option. For processors with a default alignment of 2, non-**char** structure members will be even-aligned. For processors with a default alignment of 4, 4-byte structure members will be quad-aligned and 2-byte structure members will be even-aligned. The default alignment for each processor is shown in Table 2-7 under the **-p***processor* option.

The **-Za2** option specifies that non-**char** structure members will be even-aligned, independent of the processor selected.

The **-Za4** option specifies that 4-byte structure members will be quad-aligned and 2-byte structure members will be even-aligned, independent of the processor selected.

The alignment options do not affect **packed** structures.

-Zm*number*

The **-Zm** option determines the granularity of the structure size.

The **-Zm** option instructs the compiler to make any **packed** structure size a multiple of *number*. By default, *number* is 1. Unpacked structures are not affected.

-Zn*number*

The **-Zn** option instructs the compiler that unpacked struct **sizeof** is a multiple of *number*.

Produce Minor Code Generation Variations (-Zx options)

> **Note**
>
> These options producing minor code generation variations, including the **-Ze**, **-Zo**, and **-Zp***n* options, appear with the **-K***x* options in the section *Produce Minor Code Generation Variations*, found earlier in this chapter.

Accept C++ Comments

-zc

-nzc (default)

Instructs the Microtec C compiler to accept C++ style comments.

Produce Function Range Information (Exception Handling)

-ze

The **-ze** option instructs the compiler driver to produce function range information for the functions in the current compilation file module. This option is required for the proper use of exception handling with the C++ compiler.

Command Line Examples

This section gives examples of the use of the compiler and driver.

For more information about the libraries described in these examples, see the section titled *Libraries* in this chapter.

Example:

```
mcc68k program.c
```

This command compiles the source file `program.c` to produce the object file **program.o** (UNIX) or **program.obj** (DOS). The object file is then linked with the default linker command file and the default library to produce the executable file **program.x** (UNIX) or **program.abs** (DOS). Finally, the object file is deleted.

Example:

```
mcc68k -g program.c
```

Like the previous example, this example compiles the C source file `program.c` to produce the executable file **program.x** (UNIX) or **program.abs** (DOS). The `-g` option specifies that debugging information be included in the executable file, which can then be debugged using the XRAY Debugger.

Example:

```
mcc68k module1.c module2.c module3.c -loutput.lst -Mcp
-Md5
```

This command compiles the three source files `module1.c`, `module2.c`, and `module3.c` to produce three object files **module1.o**, **module2.o**, and **module3.o**. A compiler-generated listing of each source file is written to the file `output.lst`. The generated code will use the PC-relative addressing mode for all code references (`-Mcp`) and the **A5**-relative addressing mode for all data references (`-Md5`). The three object files are then linked with the default linker command file and library to produce the executable file **module1.x** (UNIX) or **module1.abs** (DOS). The name of the executable file is derived from the name of the first source file that appears on the command line; the default extension is added to it. The object files are not deleted in this case; an object file is deleted only if a single source file is successfully

linked.

Example:

```
mcc68k -c module.s
```

This command assembles the assembly file `module.s` to produce the object file **module.o** (UNIX) or **module.obj** (DOS). The `-c` option prevents the driver from linking this module to produce an executable file.

Example:

```
mcc68k main.c asmfile.s extra.o utilities.lib
```

This command performs the following actions:

- Compiles `main.c` to produce **main.o** (UNIX) or **main.obj** (DOS).
- Assembles `asmfile.s` to produce **asmfile.o** (UNIX) or **asmfile.obj** (DOS).
- Links the object files for `main.c` and `asmfile.s` with `extra.o` and with the default linker command file and the default library to produce an executable file named **main.x** (UNIX) or **main.abs** (DOS). Modules from `utilities.lib` are also linked in if necessary.

Example:

```
mcc68k -Wa,-l>main.l main.c
```

This command compiles `main.c` to produce **main.o** (UNIX) or **main.obj** (DOS). The `-Wa` option causes the compiler to pass the subsequent `-l` option to the assembler, which generates a listing file named `main.l`.

Example:

```
mcc68k -Wl,-m>main.map main.c
```

This command compiles `main.c` to produce **main.x** (UNIX) or **main.abs** (DOS). The `-Wl` option causes the compiler to pass the `-m` option to the linker to instruct it to generate a map file named `main.map`.

Using Options Wisely

This section discusses how to use some of the compiler options to get the most out of the Microtec ANSI C compiler.

Option Combinations

Some common combinations of options are shown in Table 2-8.

Table 2-8. Common Option Combinations

Options	Result
-ng -nOg	Fastest compilation (no debugging information, no global optimization)
-O -Ot	Fastest generated code (advanced optimizations, optimize for time instead of space)

(cont.)

Table 2-8. Common Option Combinations (cont.)

Options	Result
-O -Os	Smallest generated code size (advanced optimizations, optimize for space instead of time)
-nx -A	ANSI-compliant code (no Microtec extensions enabled, only ANSI extensions enabled)
-x -A	Broadest C language definition accepted (both Microtec and ANSI extensions enabled)

ANSI Extensions

By default, the compiler enables both the ANSI (**-A**) and Microtec extensions (**-x**). When the ANSI compiler option is disabled, the compiler attempts to behave as much as possible like a traditional C compiler (K & R). However, the preprocessor is ANSI in nature, so its evaluation mechanisms are quite different from pre-ANSI preprocessors.

Using __STDC__

The **__STDC__** macro is defined by all ANSI-compliant compilers to indicate that the ANSI Standard C language definition is being enforced. Use of the **-nA** option undefines this macro.

The **__STDC__** macro lets you identify sections of your code so that different actions can be performed based upon whether ANSI checking is enforced.

Example:

```
#ifdef __STDC__
     int other_name;
#else
     int const;
#endif
```

This example declares a variable named `const` when ANSI rules are not in effect. When ANSI rules are applied (`__STDC__`), `const` is recognized as a keyword, so the variable cannot be named `const`.

Function Prototyping

Function prototypes let you include parameter type information within C function declarations. Having access to such information, the C compiler can detect mismatched parameter passing in terms of parameter type, size, and number. This

optional enforcement of parameter type checking detects some of the most common C programming errors during compilation and helps create problem-free code.

Standard library headers include ANSI and pre-ANSI declarations, which are selectively included by the setting of the ANSI compiler option. This mechanism can be copied if you do not want to give up ANSI features or lose portability with pre-ANSI compilers.

Example:

```
#ifdef __STDC__
extern int remove (const char *filename);
#else
extern int remove ();
#endif
```

Function prototypes are actually accepted regardless of the use of the ANSI compiler option. When the compiler is invoked while specifying that ANSI not be enforced, prototype information is flagged with an informational **(I)** diagnostic message.

ANSI and Floating-Point Variables

If the compiler is invoked with the ANSI compiler option, **float** variables are not converted to **double** unless necessary.

Example:

```
#ifdef __STDC__
     assert(sizeof(float)==4);
#else
     assert(sizeof(float)==sizeof(double));
#endif
```

This example shows the ANSI C definition (**__STDC__**) enforcing **float** values to be 32 bits. If the assertion is untrue (i.e., **float** values are not implemented as 32 bits), the program terminates.

If you are migrating to an ANSI compiler, be careful when there is no floating-point unit (FPU) available and software floating-point libraries are used. By using **float** instead of **double** (using the faster 32 bits instead of 64 bits), efficiency can be improved because the compiler assists you in modifying programs for maximum efficiency.

Example:

```
1     #pragma option -nQ
2     float a,b;
3     foo()
4     {
5           a = b+2.0;
                  ^
>> (I) [ANSI] inefficient evaluation in "double" because of
"double" constant
6     }

1 Informational
```

The informational message shown in this example indicates that the **double** operand (i.e., the constant `2.0`) inhibited the use of simpler floating-point arithmetic. The variable `b` will be promoted to `double` for the purposes of evaluation, which may generate less than optimal code. You can modify the source file to impose the type **float** on the constant by either using the ANSI qualifier **f** or by casting:

```
a = b+2.0f;       /* ANSI suffix denotes a float constant */
a = b+(float)2.0;/* casting of the constant to float */
```

Microtec Extensions

The **-x** option activates extensions to the C language that Microtec has implemented to ease embedded and systems programming. These extensions include:

- The **asm()** pseudofunction to in-line assembly code
- The **interrupt** keyword for interrupt procedures
- The **packed** keyword to avoid padding in structures and reduce the size of small enumerations
- The support of **char**, **short**, and **long** bit fields
- In-lining of several library functions (see the **-x** option in section *Command Line Options* in this chapter for more information)

Table 2-9 lists the additional keywords recognized by the Microtec C compiler regardless of whether the **-x** option is enabled. These keywords have been added by Microtec to implement extensions to the C compiler. Other identifiers cannot conflict with them.

Table 2-9. Keywords (Microtec Extensions)

Keyword	Description
interrupt	A function can be declared as an interrupt handler by using the **interrupt** keyword.
packed	The **packed** keyword is used in **struct**, **union**, and **enum** declaration statements to minimize the amount of storage needed by forcing no padding between members.
typeof	The **typeof** operator determines the data type of a variable.
unpacked	The **unpacked** keyword is used in **struct**, **union**, and **enum** declaration statements to force normal padding between members.

typeof() Operator

The **typeof()** operator is a Microtec extension that can be used to find the type of any variable. It can be used anywhere a type is needed. For example, the following macro can be used to swap two variables of the same type.

```
#define swap(x,y) do { \
      typeof(x) tmp = x; \
      x = y; \
      y = tmp; \
} while (0)

int a, b;
struct foo c, d;

main()
{
      swap(a, b);
      swap(c, d);
}
```

Exception Handling

When you use C programs that may eventually be used in conjunction with C++ programs using exception handling, the exception handling features of the compiler must be enabled. In this case, you must compile the C files using the **-ze** command line option. This option instructs the C++ driver to produce function range information for the C files compiled; this information is required by the exception handling run-time system.

An application can easily contain C routines. This option is introduced mainly for handling exceptions in an application of mixed C and C++ code. This option must be used to compile the routines that may be involved in a C++ application with exception handling. When the **-ze** option is specified, the C compiler (invoked by the C++ driver) emits program counter (PC) range information for functions in the given C file. The C++ exception handling system uses this information to handle exceptions. For further information on C++ exception handling, see the Microtec *C++ Compiler User's Guide and Reference for the 68000 Family*.

Customizing the Compilation Driver

The compiler features a powerful compile/assemble/link driver that automatically invokes the appropriate product. In addition to simplifying compiler invocation, the driver can be used to customize the compilation environment for your development needs.

Compiler options are read from left to right. If an option is specified multiple times on the command line, the last occurrence of that option is enforced. Options specified apply to all files being compiled (the position of file names on the command line has no significance).

Example:

```
mcc68k -g -l -Flp44 main.c read.s scr.lib -ng -Flp100 -g -ng
```

This example compiles `main.c` without debugging information (the rightmost option is `-ng`) and a listing page length of 100 lines (the rightmost option is `-Flp100`).

Example:

```
#!/bin/csh
set extraoptions="$argv[2-$#argv]"
mcc68k $1 -c -g -O -Flp44 $extraoptions
```

This UNIX C shell script, named **compile**, assigns all the invocation arguments that follow the first argument (the file name) to `extraoptions`:

`compile test.c`	Compiles `test.c` with normal options
`compile test.c -g -l`	Compiles `test.c` with normal options plus `-g` and `-l` options
`compile test.c trick.c`	Compiles `test.c` and `trick.c` with normal options

If you are not satisfied with the default setting of some of the options, you can use the driver option **-d***filename* to customize your compilation. This option also lets a system administrator establish installation defaults.

The driver reads the lines in *filename* as though they were specified on the command line. Multiple lines can be in the file; the newline character is interpreted as white space.

Example:

```
-g
-Flp66
-e /usr/misc/myproject_linkfile
-DDEBUG_MODE -DHOST=SUN4
```

If the file **/usr/misc/myproject_defaults** contained the lines shown in this example, all these options would be applied to your compilation if you specified **-d/usr/misc/myproject_defaults** on the command line. Alternatively, you could use a shell **alias** command to ensure your customization is in force:

```
alias mcc68k "mcc68k -d/usr/misc/myproject_defaults"
```

Pragmas and Options

By using the **#pragma option** ANSI construct, options normally set on the command line may be set within the program file itself. This directive lets you specify local compiler options inside the source file. These options will override those on the invocation line (since they are the last encountered) and affect the compilation of the entire file, not just code subsequent to their occurrence.

Example:

```
#pragma option -g
funct1()          /* always include debugging info */
...
#pragma option -ng
funct2()          /* do not include debugging info */
...
```

If the lines shown in this example occur in the same file, the compiler will apply the last option encountered (`-ng`) to the entire file. The compiler will not apply different options for `funct1` and `funct2`. In other words, the options cannot be turned on and off to only affect portions of a source file.

A typical use of **#pragma option** is with makefiles that compile a related group of sources. One of the files in this group may require particular options, such as opti-

mizing for space or eliminating debugging information. The **#pragma option** directive lets you specify command line options that apply to that file only:

```
#pragma option -ng      /* No debugging info for this file */
```

Only true compiler options are accepted. Driver options such as **-S**, **-e**, or **-l** are not accepted since the driver decides how to proceed before reading any input files, and all driver options have been processed prior to reading your file.

Example:

modulea.c:

```
#pragma option -NTcode_a -NIdata_a -NCdata_a -NSdata_a
#pragma option -NLdata_a
```

moduleb.c:

```
#pragma option -NTcode_b -NIdata_b -NCdata_b -NSdata_b
#pragma option -NLdata_b
```

This example shows two separate modules with the **#pragma option** directive declaring different names for compiler-generated sections within the different modules. By specifying this information in the separate modules, the sections can be located separately at link time.

Locating Header Files

Sometimes you will need to know the exact origin of header files included during a compilation. The quickest way to access this information is to use the **-E** compiler option, which invokes the preprocessor and produces an annotated listing to the standard output device.

Example:

```
#line 1 "input.c"
#line 1 "./x.h"

     extern foo(struct tag *p);

#line 2 "input.c"
#line 1 "/usr/mri/include/mcc68k/stdio.h"

     extern struct iobuf {
     ...
```

This example shows preprocessor output from the command line:

```
mcc68k -E input.c
```

for a file `input.c` containing:

```
#include "x.h"
#include <stdio.h>
```

By viewing the full path names of the include files used, you will eliminate any uncertainty about the actual header files used with your program. Refer to the **-J** compiler option for information on overriding the standard include file (**<>**) location.

Finding Source Files

It is very common to keep sources and objects in the same directory. However, it is also common to want different object versions of those sources. These two opposing desires can be accommodated with the Microtec compiler.

Example:

```
cd /mydir/no_debug_dir
mcc68k -ng /mydir/new/*.c
cd /mydir/debug_dir
mcc68k -Gf -g /mydir/new/*.c
```

This example shows how you can leave your source files in one directory and create special object versions of those files in different directories. In this case, the source files are located in the directory `/mydir/new`. Before compiling, two different subdirectories were created: `/mydir/debug_dir` and `/mydir/no_debug_dir`. Once you move to the appropriate subdirectory, you can invoke the compiler with the path name to the source files.

To generate object files that do not contain debugging information, the `-ng` option is specified with the path name to the source files. The resulting object files are placed into the current directory `/mydir/no_debug_dir`.

To generate object files that contain debugging information, the `-g` and `-Gf` options are specified with the path name to the source files. The `-Gf` option places full path names in the debug file, making it possible for the XRAY Debugger to locate the source files. The resulting object files are placed into the current directory `/mydir/debug_dir`.

In conclusion, you end up with one set of object files that do not contain debugging information (`/mydir/no_debug_dir`), another set of object files that contain debugging information (`/mydir/debug_dir`), and your source files, which remain in their original directory (`/mydir/new`).

Determining Option Defaults

The options that you specify for a given compilation only represent a subset of all the options applied to your file(s). Many compiler options are turned on by default, and you should be aware of how these defaults can affect your compilation.

An easy technique for determining the options in effect is to compile an empty file and produce a listing file:

```
mcc68k -l -y empty.c
```

All of the options in force will be listed. The `-y` option limits the compiler to checking the syntax of the program so that errors produced by an empty file are kept to a minimum.

Producing Listing Files

The listing files produced by the compiler are intended for line printer output with a default page length of 55 lines. You can change this number with the **-Flp***number* option. Specifying **-Flp0** tells the compiler to prepare the listing for a continuous printout with no page breaks.

The **#pragma list** directives can turn the listing option on and off. These directives apply to code following the directive so you can turn the listing option on and off several times within the same file.

Example:

```
1
2     foo()
3     {
4     #pragma list off
5           int invisible, visible;

6           invisible=1;
7     #pragma list on
8           visible=99;
9     #pragma list resume
10          invisible=1;
11    #pragma list resume
12          visible=99;
13    }
```

The listing produced from this example is:

```
1
2     foo()
3     {
4     #pragma list off
8           visible=99;
9     #pragma list resume
12          visible=99;
13    }
```

The **#pragma list resume** directive restores the status of the listing to the setting that was in force prior to its previous matching **#pragma list off** or **#pragma list on** directive. The listing option **-l** must be specified in order for the **#pragma list** directives to have any effect.

To produce a listing file that contains both assembly and C code, use the **-Fsm** compiler option (C code appears in the assembly source file as comment lines):

```
mcc68k -S -Fsm hello.c
```

Libraries

This section describes the libraries distributed with MCC68K and how they are used.

Distribution Files

The libraries in the distribution are collectively referred to as **mcc68k*xx*.lib** in all MCC68K documentation.

The driver will automatically select the correct library to pass to the linker if you are using the default linker command file. If you are using a different linker command file or if you invoke the linker directly, you must pass the correct library to the linker yourself.

To determine which library to use, see Table 2-10. The compiler generates absolute code and data by default; therefore, if defaults are used, use the library **mcc68kab.lib** on UNIX or **68000\mcc68kab.lib** on DOS. However, if you specify command line options that produce position-independent data references, you must use the appropriate library. For example, if you compile with the **-Mdp** option, which generates data that is position-independent relative to the program counter, you should link in the library **mcc68kpc.lib** on UNIX or **68000\mcc68kpc.lib** on DOS. The correct library also depends on the processor you specified with the **-p** option when your source was compiled (see Table 2-7 under the **-p** option).

Table 2-10. UNIX/DOS Library Use Conditions

When the Compiler Generates:	Use Library: (UNIX)	Use Library: (DOS)
68000 instructions with the options **-Mca**, **-Mda** (default)	mcc68kab.lib	68000\mcc68kab.lib
68000 instructions with the options **-Mcp**, **-Mdp**	mcc68kpc.lib	68000\mcc68kpc.lib
68000 instructions with the options **-Mcp**, **-Md5**	mcc68ka5.lib	68000\mcc68ka5.lib
68000 instructions with the options **-Mca**, **-Mda**, and **-f**	mcc68kabf.lib	68000f\mcc68kab.lib
68000 instructions with the options **-Mcp**, **-Mdp**, and **-f**	mcc68kpcf.lib	68000f\mcc68kpc.lib
68000 instructions with the options **-Mcp**, **-Md5**, and **-f**	mcc68ka5f.lib	68000f\mcc68ka5.lib
CPU32 instructions with the options **-Mca**, **-Mda** (default)	mcc68k32ab.lib	cpu32\mcc68kab.lib
CPU32 instructions with the options **-Mcp**, **-Mdp**	mcc68k32pc.lib	cpu32\mcc68kpc.lib
CPU32 instructions with the options **-Mcp**, **-Md5**	mcc68k32a5.lib	cpu32\mcc68ka5.lib
CPU32 instructions with the options **-Mca**, **-Mda**, and **-f**	mcc68kab32f.lib	cpu32f\mcc68kab.lib
CPU32 instructions with the options **-Mcp**, **-Mdp**, and **-f**	mcc68kpc32f.lib	cpu32f\mcc68kpc.lib
CPU32 instructions with the options **-Mcp**, **-Md5**, and **-f**	mcc68ka532f.lib	cpu32f\mcc68ka5.lib
68020, 68EC020, 68030, 680EC030, 68EC040, 68EC060, CPU32+ instructions with the options **-Mca**, -**Mda** (default)	mcc68kab020.lib	68020\mcc68kab.lib

(cont.)

Table 2-10. UNIX/DOS Library Use Conditions (cont.)

When the Compiler Generates:	Use Library: (UNIX)	Use Library: (DOS)
68020, 68EC020, 68030, 680EC030, 68EC040, 68EC060, CPU32+ instructions with the options **-Mcp**, **-Mdp**	mcc68k020pc.lib	68020\mcc68kpc.lib
68020, 68EC020, 68030, 680EC030, 68EC040, 68EC060, CPU32+ instructions with the options **-Mcp**, **-Md5**	mcc68k020a5.lib	68020\mcc68ka5.lib
68020, 68EC020, 68030, 680EC030, 68EC040, 68EC060, CPU32+ instructions with the options **-Mca**, -**Mda**, and **-f**	mcc68kab020f.lib	68020f\mcc68kab.lib
68020, 68EC020, 68030, 680EC030, 68EC040, 68EC060, CPU32+ instructions with the options **-Mcp**, **-Mdp**, and **-f**	mcc68kpc020f.lib	68020f\mcc68kpc.lib
68020, 68EC020, 68030, 680EC030, 68EC040, 68EC060, CPU32+ instructions with the options **-Mcp**, **-Md5**, and **-f**	mcc68ka5020f.lib	68020f\mcc68ka5.lib
68040, 68060 instructions with the options **-Mca**, -**Mda** (default)	mcc68kab040.lib	68040\mcc68kab.lib
68040, 68060 instructions with the options **-Mcp**, **-Mdp**	mcc68kpc040.lib	68040\mcc68kpc.lib
68040, 68060 instructions with the options **-Mcp**, **-Md5**	mcc68ka5040.lib	68040\mcc68ka5.lib

The libraries should be used under the conditions shown in Table 2-11.

Table 2-11. Library Use Conditions

Library	Condition
ab	When generating absolute data references (**-Mda**)
f	When generating code which uses 68881 instructions (**-f**)
pc	When generating position-independent data references relative to the program counter (**-Mdp**)
a5	When generating position-independent data references relative to the **A5** register (**-Md5**)
32	When generating CPU32 code (see the **-p***processor* option)
020	When generating code for the 68020, 68030, 68EC040, 68EC060, and CPU32+ processor (see the **-p***processor* option)
040	When generating 68040 or 68060 code (**-p68040 | -p68060**)

Refer to the section *Position-Independent Code and Data* in Chapter 9, *Embedded Environments*, for more information about position-independent code.

If you want to replace standard library routines with your own custom routines, load your routines before loading the standard libraries.

The C Library

The C libraries supplied with MCC68K provide the commonly used C language functions. See Chapter 5, *Library Extensions*, for a list of all the Microtec library extensions provided.

UNIX-Style System Functions

Certain primitive functions are highly system-specific and cannot be adequately implemented in the C library. These functions are mainly I/O related, and the operating system must provide them. These functions are collectively referred to as the UNIX-style system functions. The MCC68K library uses the UNIX-style system functions listed in Table 2-12. Although the **ftell** function is not a UNIX system function, it is used by the MCC68K library.

Table 2-12. UNIX-Style System Functions Used by the MCC68K Library

Function	Description
close	Closes a specified file.
creat	Creates a specified file.
_exit	Terminates a program.
ftell	Gets the current location in a file.
lseek	Sets the current location in a file.
open	Opens a specified file.
read	Reads bytes from a specified file.
sbrk	Allocates memory space.
unlink	Unlinks a file name.
write	Writes bytes to a specified file.

This toolkit is designed for developing programs that run in an embedded system that may not use an operating system. You may need to replace some or all of the UNIX-style system functions in order to successfully run programs in an embedded environment. Refer to section *User-Modified Routines* in Chapter 9, *Embedded Environments*, for more information on providing UNIX-style system support for the embedded environment.

Start-Up Routine

A start-up module is also included in the library. This sample routine sets up an environment before the **main** function begins execution. The start-up routine serves as the program's entry point and is responsible for a number of tasks, including opening standard files, building **argv** (the argument vector), and calling the **main** function.

If you are developing a program that will execute in an embedded system, you may need to write a special start-up routine or modify the sample start-up routine provided with your distribution files for your particular environment. Refer to your *Release Notes* for information about distribution files.

Refer to section *User-Modified Routines* in Chapter 9, *Embedded Environments*, for more information on using a start-up module in an embedded environment. By default, the compiler will use the start-up module in the C library. If you wish to

replace the standard start-up module with your own version, load your version (using the **-e** command line option) prior to loading the standard library.

Generating an Executable Program

The MCC68K Compiler allows you to take a C source file and produce an absolute object module. The following sections describe how to do this for the sample program **sieve.c**. Follow these steps:

1. Compile the sample program by entering:

```
mcc68k -c -g -o sieve.o sieve.c
```

   This command compiles `sieve.c` and produces the relocatable object module `sieve.o`. The `-g` option tells the compiler to include debugging information in module `sieve.o`. The `-c` option prevents linking (used here for the purpose of illustration).

> **Note**
>
> This example uses the **-c** compiler option to prevent linking so that all compilation steps can be shown. Normally, the command line:
>
> ```
> mcc68k -g sieve.c
> ```
>
> would be used to create an executable file containing debugging information. The assembler and linker are automatically invoked by the compilation driver.
>
> If you want to invoke the assembler or linker directly, you can use the **-Vd** or **-Vi** compiler options to examine the commands that the driver uses to invoke these tools. These commands can then be used as a guide for constructing your own invocation commands.

2. The object module automatically created by the assembler must now be linked with the library files and converted into absolute format. Microtec provides a default linker command file called **mcc68k.cmd**.

Link the object files by entering:

```
mcc68k sieve.o -o sieve.abs -Wl,"-m>sieve.map"
```

The compilation driver invokes the linker, links the appropriate libraries, etc., and creates the absolute object module `sieve.abs`. The linker command (`-m`) generates a link map and places it in the file `sieve.map`.

This step is unnecessary if the `-c` compiler option is not used (see the section *Invoking the Linker* in this chapter).

The absolute file `sieve.abs` is ready to be debugged using the XRAY Debugger or downloaded to an in-circuit emulator or a target system.

Note

Create linker command files for your applications based on the **mcc68k.cmd** default command file provided with your distribution files.

Invoking the Linker

Unless the **-c** option is entered, the driver automatically invokes the linker. When the linker is invoked, the driver passes it the appropriate library and the default linker command file **mcc68k.cmd**. You can choose an alternate linker command file with the **-e** compiler option (see the section *Pass Command File to Linker (Driver Option)* in this chapter).

The driver selects which library to pass to the linker based on the **-p** and **-Md** options. It is important that the same settings for these options are used on all driver

invocations to produce a given executable.
For more information on the LNK68K Linker, see the *Assembler/Linker/Librarian User's Guide and Reference for the 68000 Family.*

The Preprocessor 3

Overview

The Microtec C compiler contains the standard ANSI C preprocessor. The preprocessor is integrated into the compiler, making the compilation process faster by requiring only one reading of the program.

The preprocessor provides predefined symbols, macros, and directives to enable you to place conditional preprocessor statements in your source code.

Predefined Symbols

The Microtec C compiler provides predefined symbols for you to do the following:

- Write code that is customized for various aspects of the application (such as data size, memory addressing, time, date, file names, and so forth)
- Write code that is customized for a target system
- Write code that is customized for a particular host cross compiler

Those command line options that initialize some of the predefined symbols are listed in the section *Compiler Options for Predefined Symbols* in this chapter.

Refer to the section *Preprocessor Directives* in this chapter for information about using the **#pragma macro** preprocessor directive to generate a list of predefined processor symbols and their values.

Basic Preprocessor Symbols

Basic predefined preprocessor symbols are shown in Table 3-1. Refer to Chapter 6, *Internal Data Representation*, in this manual for more information about data types and storage layout.

Table 3-1. Basic Preprocessor Symbols

Symbol	Description
_BIG_ENDIAN	This symbol is set to 1 if the least significant byte of a data object is stored at a base memory address and more significant bytes are stored at successively higher memory addresses. (default: 1 for the 68000 family)
_ _DATE_ _	The value of _ _**DATE**_ _ is a character string that consists of the ASCII representation of the current date. It has the following format: *Mmm dd yyyy* *Mmm* has the same format as the month given by the **asctime** run-time function. *dd* has a leading blank if it is a one-digit number. For example, 5 May 1989 is represented as: May 5 1989 (default: current date)
_ _FILE_ _	The value of _ _**FILE**_ _ is a character string that consists of the ASCII representation of the current file name. Use the **#line** directive to change the value of _ _**FILE**_ _. (default: current file name including suffix)
_ _LINE_ _	The value of _ _**LINE**_ _ is a decimal number stored as an **int** that represents the current line number within the current file. Use the **#line** directive to change the value of _ _**LINE**_ _. (default: current line number)
_LITTLE_ENDIAN	This symbol is set to 1 if the most significant byte of a data object is stored at a base memory address and less significant bytes are stored at successively higher memory addresses. (default: undefined)
_MCC	This symbol is always defined as 0. (default: 0)
_MRI	This symbol is set to 1. (default: 1)

(cont.)

Table 3-1. Basic Preprocessor Symbols (cont.)

Symbol	Description
_MICROTEC	This symbol is set to 1. (default: 1)
_PACKED_STRUCTS	This symbol is set to 1 if the keyword **packed** is enabled. (default: 1 for the 68000 family)
_PTRDIFF_T	The type of the result of subtracting two pointers. (default: **signed int**)
_SIZE_T	The type of the result of the **sizeof()** operator. (default: **unsigned int**)
_SIZEOF_CHAR	The size of a character in bytes. (default: 1 byte for the 68000 family)
_SIZEOF_CODE_POINTER	For large memory models: the size of the **far** pointer. For all other models: the size of a code pointer. (default: 4 bytes for the 68000 family)
_SIZEOF_DATA_POINTER	For large memory models: the size of the **far** pointer. For all other models: the size of a data pointer. (default: 4 bytes for the 68000 family)
_SIZEOF_DOUBLE	The size of a **double** in bytes. (default: 8 bytes for the 68000 family or 4 bytes with the **-Kq** option enabled)
_SIZEOF_FLOAT	The size of a **float** in bytes. (default: 4 bytes for the 68000 family)
_SIZEOF_INT	The size of an **int** in bytes. (default: 4 bytes for the 68000 family)
_SIZEOF_LONG	The size of a **long** in bytes. (default: 4 bytes for the 68000 family)

(cont.)

Table 3-1. Basic Preprocessor Symbols (cont.)

Symbol	Description
_SIZEOF_LONG_DOUBLE	The size of a **long double** in bytes. (default: 8 bytes for the 68000 family or 4 bytes with the **-Kq** option enabled)
_SIZEOF_POINTER	The size of a pointer in bytes. (default: 4 bytes for the 68000 family)
_SIZEOF_SHORT	The size of a **short** in bytes. (default: 2 bytes for the 68000 family)
_STACK_GROWS_UP	This symbol is always defined as 0, indicating that the stack does not grow towards increasing addresses. (default: 0)
_STACK_GROWS_DOWN	This symbol is always defined as 1, indicating that the stack does grow towards decreasing addresses. (default: 1)
__STDC__	The value of **__STDC__** is the decimal constant 1. When this symbol is set to 1 using the **-A** option, the compiler will compile the program for ANSI C. (default: 1)
__TIME__	The value of **__TIME__** is a character string that consists of the ASCII representation of the current time in 24-hour clock format. It has the following format: *hh:mm:ss* For example, "22:30:45" is 10:30:45 P.M. (default: current time)
_VERSION	The version of the compiler as a string. (default: current version)
_WCHAR_T	The type of a wide character constant. (default: unsigned **char**)

Example:

```
#if _SIZEOF_INT == 4
#error "this program is not suitable for 16-bit targets"
#endif
```

This example checks to see if the size of an integer is 4 bytes. If an integer is 4 bytes (32 bits), then an error message is displayed using the `#error` directive.

Example:

```
#if _STR_CMP(_VERSION, "4.0") >= 0
     version = 4;
#else
     version = 3;
#endif
```

The `_STR_CMP` macro evaluates `_VERSION` and compares the result with the string `4.0`. If it matches, zero is returned; if it is greater than `4.0`, a positive number is returned. Refer to section *Preprocessor Macros* in this chapter for more information about the `_STR_CMP` macro.

Example:

```
#ifdef _BIG_ENDIAN
     printf("Lower addresses => higher order bytes");
#else
     printf("Lower addresses => lower order bytes");
#endif
```

The **_BIG_ENDIAN** or **_LITTLE_ENDIAN** preprocessor symbol is set based upon the addressing style of your target processor. If lower addresses correspond to higher order bytes, the **_BIG_ENDIAN** preprocessor symbol is set to 1. See Chapter 6, *Internal Data Representation*, in this manual for more information about storage layout.

Preprocessor Symbols for Targets

Use the predefined preprocessor symbols shown in Table 3-2 to customize target system-dependent code.

Table 3-2. Preprocessor Symbols for Targets

Symbol	Description
_MCC68K M68000 MC68000 M68K	Motorola 68000 family (defined as 1)

Example:

```
#if _MCC68K
     printf("host compiler is MCC68K");
#else
     printf("host compiler is not MCC68K");
#endif
```

Preprocessor Symbols for Hosts

Some preprocessor symbols are predefined (set to 1) only if a program is compiled on a certain host system. Otherwise, they are undefined. You can use the symbols shown in Table 3-3 to customize host compiler code.

Table 3-3. Preprocessor Symbols for Hosts

Symbol	Description
_HP9000_700	Hewlett-Packard HP 9000 Series 700 computers
_H7U	
_MSDOS	Any host using MS-DOS or PC-DOS
_MSD	
_PC	PC and compatible computers using MS-DOS or PC-DOS
_SUN4	Sun Microsystems SPARCstation computers using SunOS
_S4U	
_UNIX	Any host using the UNIX operating system

Compiler Options for Predefined Symbols

Predefined preprocessor symbols are set to 1 if enabled (except the symbols **_PIC_REG**, which contains the number of the register used for register-relative code addressing, and **_PID_REG**, which contains the number of the register used for register-relative data addressing) or set to 0 if otherwise. Table 3-4 lists these preprocessor symbols and their related compiler options.

Table 3-4. Preprocessor Symbols and Corresponding Compiler Options

Preprocessor Symbol	UNIX/DOS Option
__STDC__[a]	-A
_CHAR_UNSIGNED	-Ku
_CHAR_SIGNED	-nKu
_DEBUG	-g
_FPU	-f
_HW_DEMANDS_ALIGNMENT	-KT
_MRI_EXTENSIONS[a]	-x
_OPTION_Kt	-Kt
_OPTION_upd	-upd
_OPTION_us	-us
_OPTION_utn	-ut*n* (returns value of *n)*
_PIC	-Mcp
_PID	-Md*n* or -Mdp
_PIC_REG	-Mcp
_PID_REG	-Md*n*
_68000	-p68000 (default)
_68008	-p68008
_68010	-p68010
_68020	-p68020
_68030	-p68030

(cont.)

Table 3-4. Preprocessor Symbols and Corresponding Compiler Options (cont.)

Preprocessor Symbol	UNIX/DOS Option
_68040	-p68040
_68060	-p68060
_68302	-p68302
_68330	-p68330
_68331	-p68331
_68332	-p68332
_68333	-p68333
_68340	-p68340
_68349	-p68349
_68360	-p68360
_68EC000	-p68EC000
_68EC020	-p68EC020
_68EC030	-p68EC030
_68EC040	-p68EC040
_68EC060	-p68EC060
_68HC000	-p68HC000
_68HC001	-p68HC001
_CPU32	-pcpu32
_CPU32P	-pcpu32p

a. These preprocessor symbols are defined only if set to 1.

Preprocessor Macros

Table 3-5 lists the predefined preprocessor macros.

Table 3-5. Predefined Preprocessor Macros

Macro Name	Description
_ _OPTION_AVAIL(*string*)	This macro checks to see if the specified command line option (*string*) is currently active. *string* must be enclosed in quotes and contain a leading dash. If the option is active, **_ _OPTION_VALUE** is replaced by 1; otherwise, it is replaced by 0.
_ _STR_CMP(*string1*, *string2*)	This macro compares its two argument strings and is replaced by the result. The strings are compared in lexicographic order (i.e., characters are compared based on their internal machine representation). For example, because an ASCII **A** is 41 hexadecimal and an ASCII **B** is 42 hexadecimal, **A** is less than **B**. **_ _STR_CMP** is replaced by the following values: • Negative integer if *string1* < *string2* • 0 if *string1* = *string2* • Positive integer if *string1* > *string2*
_STR_CMP(*string1*, *string2*)	This macro is identical to **_ _STR_CMP** except that it can evaluate arguments.
_ _STR_CASE_CMP(*string1*, *string2*)	This macro is identical to the **_ _STR_CMP** macro except that it is case-insensitive.
_STR_CASE_CMP(*string1*, *string2*)	This macro is identical to the **_ _STR_CASE_CMP** macro except that it can evaluate arguments.
_ _STRIP_QUOTES(*string*)	This macro takes a single argument that is a string literal and removes the quotes.

Example:

```
#if __OPTION_VALUE("-g")
```

The `-g` compiler option generates debugging information for the XRAY Debugger. By using the `__OPTION_VALUE` macro, you can exercise different portions of your code when the debugging option is turned on or off.

Example:

```
__STR_CMP("hella","hello")
```

The __STR_CMP macro is replaced by -14 (the difference between ASCII “a” and “o”).

Example:

```
#if !_STR_CMP(__FILE__, "main.c")
main()
     printf("Only compiled if source file named main.c\n");
}
#endif
```

If the __FILE__ macro evaluates to the string main.c, the value 0 is returned. Note that the **__STR_CMP** macro cannot perform this operation since it cannot evaluate arguments.

Example:

```
__STR_CMP("hella","hellA")
__STR_CASE_CMP("hella","hellA")
```

The __STR_CASE_CMP macro will show a successful comparison (0) while the __STR_CMP macro will not.

Example:

```
#if !_STR_CASE_CMP(__FILE__, "hELLo.c")
main()
     printf("Compiled if source file named hello.c\n");
}
#endif
```

The __STR_CASE_CMP macro will show a successful comparison (0). The main function will be executed if the source file is named hello.c with any combination of capitalization (e.g., hello.c, Hello.c, heLLo.c, HELLO.C, etc.). Note that the **__STR_CASE_CMP** macro cannot perform this operation since it cannot evaluate arguments.

Example:

```
int __STRIP_QUOTES("a")
```

The __STRIP_QUOTES macro removes the double quotes (") that form the two ends of the string literal "a" and leaves the ASCII character a, which has a decimal value of 97. The __STRIP_QUOTES macro is useful for using the value returned by some preprocessor symbols.

The **_PID_REG** and **_PIC_REG** preprocessor symbols always return a quoted string. Thus, it might be necessary to the `__STRIP_QUOTES` macro to remove the quotes.

Example:

```
/*a version of __STRIP_QUOTES which evaluates its arg
first*/
#define _STRIP_QUOTES(__x)__STRIP_QUOTES(__x)

_PID_REG                        may yield "A5"
_STRIP_QUOTES(_PID_REG)         may yield A5
```

Preprocessor Directives

In addition to the standard directives provided by the ANSI C preprocessor, MCC68K provides several directives that extend beyond standard ANSI C. This section provides a syntax, description, and example of these preprocessor directives. Table 3-6 lists the Microtec preprocessor directives.

Table 3-6. Microtec Preprocessor Directives

Directive	Description
#informing	Issues an informational message.
#pragma asm	Begins embedded assembly instructions.
#pragma eject	Controls page listing.
#pragma endasm	Ends embedded assembly instructions.
#pragma error	Issues an error message and halts compilation.
#pragma info	Issues an informational message.
#pragma list	Controls source line listings.
#pragma macro	Lists predefined processor symbols.
#pragma option	Specifies command line options.
#pragma warn	Issues a warning message.
#warning	Issues a warning message.

#informing — Issues an Informational Message

Syntax

```
#info message
#inform message
#informing message
```

Description

message — An informational message. No quotes are required around *message*.

The **#informing** directive displays an informational message on the standard output device. After the message is displayed, compilation continues.

Notes

None.

Example

```
#if VERSION < 3
#info Not Valid Version Number
#endif
```

#pragma asm — Begins Embedded Assembly Instructions

Syntax

```
#pragma asm
```

Description

The **#pragma asm** directive indicates that the following source lines are to be interpreted as assembly code. The **#pragma endasm** directive indicates the end of the assembly code sequence.

Use the **#pragma asm** and **#pragma endasm** directives only outside of a function body.

Notes

You can use the **asm** pseudofunction to embed assembly code within a function body. See *In-Line Assembly* in Chapter 9, *Embedded Environments*, in this manual for more information.

Macros and preprocessor directives can be used within the assembly code.

Example

```
#pragma asm
        (assembly source code)
#pragma endasm
```

#pragma eject — Controls Page Listing

Syntax

```
#pragma eject
```

Description

The **#pragma eject** directive forces page ejects in listing output. When the **#pragma eject** command is used while listing output, the compiler will cease the output listing on the current page, eject the page, then continue listing output at the beginning of the next page.

Notes

None.

Example

```
#pragma eject
```

#pragma endasm — Ends Embedded Assembly Instructions

Syntax

```
#pragma endasm
```

Description

The **#pragma endasm** directive indicates the end of the assembly code sequence.

Use the **#pragma asm** and **#pragma endasm** directives only outside of a function body.

Notes

You can use the **asm** pseudofunction to embed assembly code within a function body. See *In-Line Assembly* in Chapter 9, *Embedded Environments*, in this manual for more information.

Macros and preprocessor directives can be used within the assembly code.

Example

```
#pragma asm
      (assembly source code)
#pragma endasm
```

#pragma error — Issues an Error Message and Halts Compilation

Syntax

```
#pragma error message
```

Description

message An error message. No quotes are required around *message*.

The **#pragma error** directive displays the message following the **#pragma error** directive on the standard output device. After the message is displayed, compilation stops.

Notes

Use this directive instead of the **#error** directive if you have disabled Microtec extensions. See the **-x** option in section *Enable Microtec Extensions* in Chapter 2, *UNIX/DOS User's Guide*, for a description of the compiler command line options that enable and disable Microtec extensions.

Example

```
#pragma option -nx
#if VERSION < 3
#pragma error Not Valid Version Number
#endif
```

#pragma info — Issues an Informational Message

Syntax

```
#pragma info message
```

Description

message An informational message. No quotes are required around *message*.

The **#pragma info** directive displays the informational message following the **#pragma info** directive on the standard output device. After the message is displayed, compilation continues.

Notes

Use this directive instead of the **#info** directive if you have disabled Microtec extensions. See the **-x** option in section *Enable Microtec Extensions* in Chapter 2, *UNIX/DOS User's Guide*, for a description of the compiler command line options that enable and disable Microtec extensions.

Example

```
#pragma option -nx
#if VERSION < 3
#pragma info Not Valid Version Number
#endif
```

#pragma list — Controls Source Line Listings

Syntax

```
#pragma list {off | on | resume}
```

Description

The **#pragma list off** directive indicates that the following source lines are not to be output to the listing file regardless of the setting of the list file command line option (see the section *Generate Listing File* in Chapter 2, *UNIX/DOS User's Guide*, for more information).

The **#pragma list on** directive indicates that the following source lines are to be output to the listing file. This directive has an effect only if the list file command line option is specified. For more information, see the section *Generate Listing File* in Chapter 2, *UNIX/DOS User's Guide*.

The **#pragma list resume** directive restores the status of the listing to the setting that was in force prior to its previous matching **#pragma list off** or **#pragma list on** directive.

Notes

None.

Example

To disable the listing of a segment of source code, enclose it with:

```
#pragma list off
      (source_code)
#pragma list resume
```

The *source_code* in the above example is not listed; there is no effect on the listing status of the surrounding code.

#pragma macro — Lists Predefined Processor Symbols

Syntax

```
#pragma macro
```

Description

The **#pragma macro** directive causes the compiler to generate a list of predefined processor symbols and their values.

Notes

None.

Example

Given the file **x.c** with the following contents:

```
#pragma macro
int dummy;
```

Use the following command line to invoke the compiler:

```
mcc68k -y x.c
```

The Microtec C compiler generates a list of predefined processor symbols and their values. The **-y** option will direct the list to the standard output device.

#pragma option — Specifies Command Line Options

Syntax

```
#pragma option [ option_list ]
```

Description

option_list Specifies compiler command line options.

The **#pragma option** directive allows command line options to be specified in the source file. Generally, only command line options related to optimization or code generation can be successfully used with this **pragma**. Options given with this **pragma** have scope over the entire module.

Options specified using **#pragma option** always take precedence over options specified on the command line.

Notes

Options for the driver or preprocessor are ignored and may cause a warning message to be given.

If the **#pragma option** directive specifies the compiler options to set ANSI-compliant mode (**-A**) or enable Microtec extensions (**-x**), the options will be ignored; these options are only effective when specified on the command line.

Example

To ensure that the module is always compiled with debugging information, use the following **#pragma** in the source code:

```
#pragma option -g
```

#pragma warn — Issues a Warning Message

Syntax

```
#pragma warn message
```

Description

message A warning message. No quotes are required around *message*.

The **#pragma warn** directive displays the warning message following the **#pragma warn** directive on the standard output device. After the message is displayed, compilation continues.

Notes

Use the **#pragma warn** directive instead of the **#warning** directive if you have disabled Microtec extensions. See the **-A** compiler option in section *Enable Microtec Extensions* in Chapter 2, *UNIX/DOS User's Guide*, for a description of the compiler command line options that enable and disable Microtec extensions.

Example

```
#pragma option -nx
#if VERSION < 3
#pragma warn Not Valid Version Number
#endif
```

#warning — Issues a Warning Message

Syntax

```
#warning message
#warn message
```

Description

message A warning message. No quotes are required around *message*.

The **#pragma warning** directive displays the warning message following the **#warning** directive on the standard output device. After the message is displayed, compilation continues.

Notes

#warning may be shortened to **#warn**.

Example

```
#if VERSION < 3
#warn Not Valid Version Number
#endif
```

C Library Customizer 4

Introduction

This chapter describes the process of customizing the libraries using the Microtec C Library Customizer. A library is a set of objects, derived from the library sources generated by a compilation. The compilation is set by the parameters of a configuration. The Microtec C compiler is distributed with a set of libraries, providing a number of general configurations for embedded systems applications. Refer to Chapter 5, *Library Extensions*, in this manual for a list of Microtec library extensions distributed with your compiler.

There are hundreds or even thousands of different ways a library module can be compiled. Furthermore, there are parts of the library that you may not want to include (e.g., the floating-point components). The Microtec C Library Customizer lets you build your own custom libraries by altering a Microtec general distribution library to suit your application.

The C Library Customizer is actually the Microtec C compiler plus certain scripts (for this chapter, the term "script" refers to UNIX shell scripts and DOS batch files).

To customize a library, copy one of the configuration files, modify it to fit your needs, and then execute a simple script. You can also validate such a library by running a test suite provided with the compiler.

When to Create a Custom Library

There are a wide variety of possible reasons to build a custom library. For example, a custom library may be needed to:

- Eliminate the linking of the floating-point package when **printf** is used (enable the definition of the preprocessor symbol **EXCLUDE_FORMAT_IO_FLOAT_FMT**)
- Debug the library by using the **-g** option
- Use the dot character instead of the underscore to prefix C names by using the **-upd** option
- Compile to take full advantage of a specific target
- Save space by excluding selected routines

- Build libraries with single-precision floating-point by using the **-Kq** option
- Rename sections to segregate library variables from program variables by using the **-N** option

Directories in the C Library Customizer

On UNIX, the following directories containing the C Library Customizer will be installed along with the Microtec C compiler:

- **cmd** — Scripts
- **src** — Source Files
- **include** — System Headers
- **test** — Test Files
- **config** — Configuration Files
- **lib** — Library Files and Linker Command Files

On UNIX, these directories will be write protected. You will need to copy the directories into your own work area before you can proceed with the building of libraries. Please contact your system administrator to obtain the installation location of the C Library Customizer.

On DOS, the C Library Customizer will be available under the directory **C:\MCC68K\RTL**. However, it is recommended that you make a working copy of the C Library Customizer in the directory in which custom libraries will be built.

All scripts are designed to run with the **cmd** directory as the default directory and they assume the given directory structure. The **cmd** directory is the location where you will probably do most of your work. The **config** directory contains all the Microtec general distribution configuration files. On DOS, the **config** directory contains additional sub-directories containing configuration files, which correspond to the libraries and library sub-directories provided on the distribution. The **config** directory is where you will place your customized configuration file.

DOS System Requirements

The C Library Customizer build and test scripts also make frequent use of environment variables. For that reason, it is recommended that you specify an environment size of at least 2048 bytes.The following is an example of how you can specify this environment size by placing a command in your **config.sys** file:

```
shell=c:\command.com /e:2048 /p
```

The actual path to your **command.com** file may vary for your system. Please refer to your DOS documentation for more information about the **shell** command.

Using the C Library Customizer

To build a customized library using the C Library Customizer, the following steps must be taken:

1. Create a custom configuration file
2. Build the library using the **bld_lib** script
3. Test the custom library using the **test_lib** and **test_one** scripts

Note

Library testing assumes the availability of the XHS68K Debugger. If you have a different debugger (monitor or emulator versions), you will need to modify the **test_lib** or **test_1.bat** scripts.

Step 1: Creating a Custom Configuration File

The first step in generating a custom library is to generate a configuration file defining that library. A configuration file is a text file that you can edit using your personal text editor. The configuration file acts as a header file that is read by the compiler and contains compiler options in the form of **#pragma option** directives. The argument to the **#pragma option** directives are actually compiler command line options (refer to section *Preprocessor Directives* in Chapter 3, *The Preprocessor*, for more information on the **#pragma option** directive). The configuration file defines how your custom library will be built.

The simplest method to create a new configuration file is to identify which existing configuration file is closest to your requirements and make a copy, giving the copied configuration file a new name. This name will also be the label of your new library.

Note

If you are using DOS, you should avoid using a name with more than 8 characters to label your new library.

To generate a configuration file, follow these steps:

1. In the **config** directory, select the general distribution configuration file which most closely matches the library you are going to build. If you want to start with a configuration file which only defines defaults, select **a_type.h**.

2. Copy the existing configuration file to a new file. The base name of the configuration file must be identical to the base name of the library you are building. So if the library is to be named *library*.**lib**, then the configuration file must be named *library*.**h**.

3. Edit your new configuration file to reflect your choice of compiler options.

There are several preprocessor symbols that may be defined to reconfigure the libraries. These symbols are also defined using the **#pragma option** directive. Currently, there are options in the configuration files that will reconfigure the formatted **printf** and **scanf** I/O routines using compiler preprocessor symbols. Enabling these symbols allows you to remove the features of these routines which you are not currently using.

Table 4-1 shows the preprocessor symbols used to customize I/O routines. These preprocessor symbols will disable options, flags, and formats for the **printf** and **scanf** library functions.

Table 4-1. C Library Customizer Preprocessor Symbols

Preprocessor Symbol	Function
Options	
EXCLUDE_FORMAT_IO_h_OPT	Disables the **h** option (indicates I/O object is a **signed** or **unsigned short int**)
EXCLUDE_FORMAT_IO_l_OPT	Disables the **l** option (indicates I/O object is **signed** or **unsigned long int**)
EXCLUDE_FORMAT_IO_L_OPT	Disables the **L** option (indicates **signed** or **unsigned long double**)
EXCLUDE_FORMAT_IO_ASSGN_SUPP	Disables the * option (allow variable precision and field width specifications) in input (**scanf**) functions

(cont.)

Table 4-1. C Library Customizer Preprocessor Symbols (cont.)

Preprocessor Symbol	Function
EXCLUDE_FORMAT_IO_STAR_OPT	Disables the * option (allow variable precision and field width specifications) in output (**printf**) functions
Flags	
EXCLUDE_FORMAT_IO_MINUS_FLAG	Disables the **-** flag (left justify)
EXCLUDE_FORMAT_IO_PLUS_FLAG	Disables the **+** flag (right justify)
EXCLUDE_FORMAT_IO_SPACE_FLAG	Disables the white space flag (causes a space to be prefixed if no **+** or **-** is indicated)
EXCLUDE_FORMAT_IO_SHARP_FLAG	Disables the **#** flag (output is to be printed in an alternate form)
EXCLUDE_FORMAT_IO_ZERO_FLAG	Disables the **0** flag (pads the field width with zeroes)
Formats	
EXCLUDE_FORMAT_IO_BRAKT_FMT	Disables the **[** format (convert character sequences)
EXCLUDE_FORMAT_IO_CHAR_FMT	Disables the **c** format (convert individual characters)
EXCLUDE_FORMAT_IO_DEC_FMT	Disables the **d** format (convert signed integer)
EXCLUDE_FORMAT_IO_INT_FMT	Disables the **i** format (convert signed integer)
EXCLUDE_FORMAT_IO_FLOAT_FMT[a]	Disables the **f** format (convert floating-point numbers)
EXCLUDE_FORMAT_IO_NUMB_FMT	Disables the **n** format (convert number of characters read or written)
EXCLUDE_FORMAT_IO_OCT_FMT	Disables the **o** format (convert octal number)
EXCLUDE_FORMAT_IO_PNT_FMT	Disables the **p** format (convert pointers)
EXCLUDE_FORMAT_IO_STR_FMT	Disables the **s** format (interpret argument as a string)

(cont.)

Table 4-1. C Library Customizer Preprocessor Symbols (cont.)

Preprocessor Symbol	Function
EXCLUDE_FORMAT_IO_UNS_FMT	Disables the **u** format (convert unsigned number)
EXCLUDE_FORMAT_IO_HEX_FMT	Disables the **x** format (convert hexadecimal number)
Line Buffered I/O	
EXCLUDE_LINE_BUFFER_DEFAULT	Disable line buffering for stdin and stdout
EXCLUDE_TERMINAL_SIMULATION	Disable character echoing during input
LINE_BUFFER_SIZE=*n*	Specifies the buffer size for I/O
Non-reentrant Functions	
EXCLUDE_ATEXIT	Disables support for the **atexit()** function.
EXCLUDE_ERRNO	Disables support for **errno** and functionswhich use **errno**.
EXCLUDE_INITDATA	Disables the call to **initcopy()**, part of the init-data feature.
EXCLUDE_IOB	Disables support for C I/O routines.
EXCLUDE_MALLOC	Disables support for all memory allocation functions and removes global data used to support them.
EXCLUDE_RAND	Disables support for **rand()** and **srand()** and removes global data used to support them.
EXCLUDE_SIGNAL_RAISE	Disables support for the **signal()** and **raise()** functions.
EXCLUDE_STRTOK	Disables support for **strtok()** and removes global data used to support it.
EXCLUDE_TIME	Disables support for all time functions and removes global data used to support them.
INCLUDE_BUILD_ARGV	Includes **build_argv()** for command line processing.

(cont.)

Table 4-1. C Library Customizer Preprocessor Symbols (cont.)

Preprocessor Symbol	Function
Math Functions	
INCLUDE_FAST_POW	Incorporates a faster, but less accurate **pow()** function.

a. If you are not using floating-point components in your program, you may enable the preprocessor symbol **EXCLUDE_FORMAT_IO_FLOAT_FMT**, which will automatically disable **%f** processing in both the **scanf** and **printf** functions. When you disable **%f** processing, the floating-point library will not be linked in and will result in a much smaller code size.

By setting the configuration file's preprocessor symbols efficiently, you can trim the formatted I/O routines down to minimal size while retaining the functionality you need.

Once you have created your configuration file, save it and move to the **cmd** directory.

Step 2: Building a Custom Library

To build a custom library, you must first make the **cmd** directory your default directory. You then execute the **bld_lib** script to create the library. The **bld_lib** script creates the library based on the contents of the configuration file.

Syntax

The following command syntax invokes the library build script for UNIX hosts:

```
bld_lib configuration_file
        [ -bindir bin_dir ]
        [ -output output_file ]
```

The following command syntax invokes the library build script for DOS hosts:

```
bld_lib subdir/configuration_file
        [ -bindir bin_dir ]
        [ -output output_file ]
```

Description

`bld_lib` Specifies the name of the library build script.

subdir Specifies the subdirectory for the library in the configuration directory (DOS only).

configuration_file Specifies the name of the configuration file. The configuration file may be stated either without a file name extension or with either a **.h** or **.lib** extension.

`-bindir` *bin_dir* Accesses Microtec C compiler, assembler, librarian, and linker from the directory *bin_dir* (UNIX only).

`-output` *output_file*
Writes the results of the build operation to *output_file* (DOS only).

The script **bld_lib** builds a custom library and places it in the **lib** directory. If the name of the configuration file was *name***.h**, then the name of the library will be *name***.lib**.

Once the **bld_lib** operation is complete, you are ready for testing.

Step 3: Testing a Custom Library

Once a library has been built, it must be tested. Two scripts are provided to help you with this process, **test_lib** and **test_one**. The **test_lib** scripts runs all available test routines. The **test_one** script, on the other hand, runs an individual test.

> **Note**
>
> To prevent compilation errors, it is recommended that you do not run the test scripts at the same time you are running the build script.

Syntax

```
test_lib configuration_file
        [ -bindir bin_dir ]
        [ -cmdfile cmd_file ]
        [ -debugger command ]
        [ -output output_file ]
        [ -testsrcfile src_file ... ]

test_one configuration_file test_file
        [ -bindir bin_dir ]
        [ -cmdfile cmd_file ]
        [ -debugger command ]
        [ -output output_file ]
        [ -testsrcfile src_file ... ]
```

Description

`test_lib` Specifies the name of the test script which runs a complete set of tests.

`test_one` Specifies the name of the test script which runs an individual test.

configuration_file Specifies the file name of the configuration file. The configuration file may be declared either without a file name extension or with either a **.h** or **.lib** extension.

test_file Specifies the name of an individual test file. This file must be located in the **test** directory.

`-bindir` *bin_dir* Accesses the Microtec C compiler, assembler, librarian, and linker from the directory *bin_dir* (UNIX only).

`-cmdfile` *cmd_file* Assigns *cmd_file* as the linker command file.

`-debugger` *command*
Assigns *command* as the XRAY Debugger name. (default: **XHS***tar* for UNIX and DOS, **X***tar* for VMS where *tar* is an abbreviation for your target processor)

`-output` *output_file*
Writes the results of the `test_lib` and `test_one` build operation to *output_file* (DOS only).

`-testsrcfile` *src_file*
Compiles and links *src_file* before the customized library. The file *src_file* will always be compiled with the full debugging option (**-g**) enabled. The file *src_file* must be located in the **src** directory.

The **test_lib** and **test_one** scripts run tests on a custom library. The script **test_lib** executes the complete test suite found in the **test** directory. The script **test_one** executes an individual test.

Upon completion, executable files generated by the compiler are placed in the **test** directory along with the output files generated by the test routines. These output files are named *test_file***.out**.

Assessing Test Results

The output of the test scripts has the following format:

- An individual test begins with the line:

  `TESTING:` *test_name* `LIBRARY:` *library_name*

  where *test_name* is the name of the test file being executed and *library_name* is the name of the library which is being tested. Compiler warnings and errors will appear after this line.

- The results of the tests appear between two lines of asterisks (*). If the test was successful, the following message is issued:

  ```
  This test has completed with no errors.
  ```

  If any other message appears or if there is any other output along with this message, the test has failed.

> **Note**
>
> If there is no output between the lines of asterisks, the test has failed.

- On UNIX systems, **test_lib** will print a summary of the test results. It will either indicate that the test suite was successful, or it will indicate that the test suite has failed and list those tests which failed. On DOS systems, **test_lib** will not generate a summary; the user must examine the results of each individual test.

Once the **test_lib** script has completed with no errors, the custom library is ready for use.

Debugging Custom Libraries

If **test_lib** indicates that there is a problem with the custom libraries, you can debug these libraries. If the custom libraries have been built with debugging information enabled, you can bring up your XRAY Debugger and step through the particular test which has failed. Since the library was built with debugging information enabled, you can easily step through the library routines.

On the other hand, if the custom library was built without debugging information enabled, you will not be able to easily step through the library routines. In this situation, you should consider taking one of the following options:

- Rebuild the library with debugging information enabled. This method is perhaps the simplest, but it does take some time to rebuild the library.

- Use the **test_one** script to recompile the test routine and those library routines where you think there might be a problem. You need to specify those library routines as arguments to the **-testsrcfile** option. The routines will be recompiled with debugging information enabled and they will be linked in ahead of your custom library. Once this is done, you can easily step through the library routines with the XRAY Debugger.

Once the problem has been located and corrected, you should rerun the individual test using the **test_one** script. If the results of the individual test indicate that the problem has been fixed, rerun the entire test suite using the **test_lib** script. When the **test_lib** script indicates that your custom library is passing all tests, the library is ready for use in your application.

Using a Custom Libraries

To use your new custom library, link your custom library as you would with any other objects (or library). However, your custom libraries must precede any general distribution libraries during the link.

If a compilation driver is available, you can specify the custom library in the command line, as in the following example:

```
mcc68k -o prog mod1.c mod2.c mod3.s mod4.o iBuiltIt.lib
```

In this example, the custom library will be passed to the linker before the general distribution libraries supplied by the driver.

If your custom library supersedes the general distribution library, you may want to use the driver to manually include the custom library name in the linker command file. You can specify your custom library by using the **-e***linker_commandfile* driver command line option, as in the following example:

```
mcc68k mod1.c -eMyCmds
```

In this mode, the driver does not supply the Microtec general distribution libraries, which can result in a faster linking. You may want to have a look at the sample linker command file provided by Microtec.

Library Extensions 5

Overview

The Microtec C compiler provides a run-time library that contains many functions familiar to C programmers. Some low-level system functions are also included with Microtec C for use on target systems; however, these functions may not be applicable to embedded systems.

The Microtec C standard include file directory is described in the Chapter 2, *UNIX/DOS User's Guide*.

C Function Groups and Include Files

In addition to the standard ANSI include files, the Microtec C compiler also provides extensions beyond the standard ANSI C function set.

mriext.h

The include file **mriext.h** includes Microtec extensions. Table 5-1 lists the functions in **mriext.h**. In Table 5-1, *c* refers to a character string.

Table 5-1. mriext.h Functions

Function	Definition
eprintf	Prints to standard error.
fileno	Gets file number.
ftoa	Converts floating-point numbers to ASCII.
getl	Reads **long int**.
getw	Reads **short int**.
isascii	Checks if argument is ASCII.
itoa	Converts integer to ASCII.
itostr	Converts **int** to ASCII, variable base.
ltoa	Converts **long** to ASCII.

(cont.)

Table 5-1. mriext.h Functions (cont.)

Function	Definition
ltostr	Converts **long** to ASCII, variable base.
max	Returns maximum value.
memccpy	Copies memory looking for a character.
memclr	Clears memory bytes.
min	Returns minimum value.
putl	Writes **long int**.
putw	Writes **short int**.
swab	Swaps bytes in memory.
toascii	Guarantees argument is ASCII.
_tolower (*c*)	Is the same as **tolower** except does not perform argument checking.
_toupper (*c*)	Is the same as **toupper** except does not perform argument checking.
zalloc	Allocates data space dynamically and initializes to zero.

The **mriext.h** file also defines the macros listed in Table 5-2.

Table 5-2. mriext.h Macros

Macro	Definition
BLKSIZE	Is the size of the I/O buffer.
FALSE	Is the value 0.
NULLPTR	Is the value of the **NULL** pointer.
stdaux	Is a pointer to a standard auxiliary output device.
stdprn	Is a pointer to a standard printer device.
TRUE	Is the value 1.

Non-Reentrant Functions and Extensions

Microtec library extensions are mostly reentrant. However, there are library extensions that are non-reentrant due to the use of static, nonautomatic data.

The Microtec non-reentrant functions and extensions are shown in Table 5-3.

Table 5-3. Non-Reentrant Functions and Extensions

Function	Static Variables Used
acos	errno
asctime	_ctbuf[ ]
asin	errno
atan2	errno
atexit	_atexit_stack[ ], _atexit_top
calloc	_membase, _avail, _badlist
clearerr	_iob[a]
cos	errno
cosh	errno
ctime	_ctbuf[ ], _xtm
eprintf	_iob[a]
exp	errno
fclose	_iob[a]
fflush	_iob[a]
fgetc	_iob [a]
fgetpos	errno[a]
fgets	_iob[a]
fmod	errno
fopen	_iob[a]
fprintf	_iob[a]

(cont.)

Table 5-3. Non-Reentrant Functions and Extensions (cont.)

Function	Static Variables Used
fputc	_iob[a]
fputs	_iob[a]
fread	_iob[a]
free	_membase, _avail, _badlist
freopen	_iob[a]
frexp	errno
fscanf	_iob[a]
fseek	_iob[a]
fsetpos	_iob[a], errno
ftell	errno[a]
fwrite	_iob[a]
getc	_iob[a]
getchar	_iob[a]
getl	_iob
gets	_iob
getw	_iob[a]
gmtime	_xtm
ldexp	errno
localtime	_xtm
log	errno
log10	errno
lseek	_iob
malloc	_membase, _avail, _badlist
modf	errno
open	_iobj

(cont.)

Table 5-3. Non-Reentrant Functions and Extensions (cont.)

Function	Static Variables Used
perror	_iob[a]
pow	errno
printf	_iob[a]
putc	_iob[a]
putchar	_iob[a]
putl	_iob[a]
puts	_iob[a]
putw	_iob[a]
raise	errno
rand	_randx
realloc	_membase, _avail, _badlist
rewind	_iob[b], errno
scanf	_iob[a]
setbuf	_iob
setvbuf	_iob
signal	errno
sin	errno
sinh	errno
sqrt	errno
srand	_randx
sscanf	_iob[a]
strtod	errno
strtok	_lastp
strtol	errno
strtoul	errno

(cont.)

Table 5-3. Non-Reentrant Functions and Extensions (cont.)

Function	Static Variables Used
tan	errno
tanh	errno
tmpfile	_iob[b]
tmpnam	_iob[b]
ungetc	_iob[a]
vfprintf	_iob[a]
vprintf	_iob[a]
zalloc	_membase, _avail, _badlist

a. This extension may have an I/O buffer attached through **_iob**. An I/O buffer is a buffer used for buffered I/O. Any I/O stream that is unbuffered will not access the buffer.

b. This library function is currently implemented as a stub which does not modify static data. However, an actual implementation of this function, which one might encounter under UNIX, would modify the described static data.

System Functions

The system functions that perform input/output operations are defined in Table 5-4. For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

Table 5-4. System Functions

Function	Definition
close[a]	Closes a specified file.
creat[a]	Creates a new file.
_exit[a]	Terminates a program.
lseek[a]	Sets the current location in a file.
open[a]	Opens a specified file.
read[a]	Reads from a specified file.

(cont.)

Table 5-4. System Functions (cont.)

Function	Definition
sbrk	Allocates memory space.
unlink[a]	Unlinks a file name.
write[a]	Writes to a specified file.

a. This function is currently implemented as a stub which does not modify static data. However, an actual implementation of this function would make this function non-reentrant and modify static variables based on your target system or kernel.

Library Function Definitions

This section describes all of the library functions in alphabetical order and presents them in a fixed format for easy reference. The format used is described below.

Function Name

Each section begins with the name of the library function followed by a short, one-line description. Note that some functions are also defined as macros.

Syntax

The syntax shows the return type of the function and the types of its formal arguments. Declarations for some of these functions are in the standard include file indicated. Alternatively, for functions which are not implemented as macros, the function can be declared as an external function with the appropriate return type.

Class

Each function is flagged as a Microtec C extension. Some functions are implemented as a macro or a system function. Functions not implemented as macros can be declared as external functions with the appropriate return type.

Description

The description gives the meaning of the function and its return values.

Notes

This section contains information on abnormal behavior, embedded environment considerations, and other issues that may affect the behavior of the function or macro.

close — Closes a Specified File

Syntax

```
int close (int fd);
```

Class

System Function

Description

The **close** function closes a file previously opened by either the **open** or **creat** function. The file descriptor *fd* is returned by **open** or **creat** when the associated file is opened. A file descriptor must be specified (as opposed to a stream address).

The **close** function returns 0 when successful or -1 if the function detects an unknown file descriptor.

Notes

For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

creat — Creates a Specified File

Syntax

```
int creat (char *filename, int mode);
```

Class

System Function

Description

The **creat** function creates and opens the file *filename* for writing. *mode* specifies the file's protection mode. For devices that cannot be created (such as terminals), the **creat** function just opens the device.

If **creat** is successful, it returns a file descriptor for *filename*. It returns -1 if the file cannot be written to, if the file is a directory, or if there are already too many files opened.

Notes

For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

eprintf — Provides Formatted Print to Standard Error

Syntax

```
#include <mriext.h>
int eprintf (const char *format, ...);
```

Class

Microtec C Extension

Description

The **eprintf** function provides formatted output to the C standard error file. The format and results are the same as for **printf**. When **eprintf** is successful, it returns the number of characters transmitted.

If an output error is encountered, **eprintf** returns **EOF** (**End-Of-File**).

Notes

The function **eprintf** modifies the static array **_iob**; no other static variables are modified.

_exit — Terminates a Program

Syntax

```
void _exit (int status);
```

Class

System Function

Description

The **_exit** function causes normal program termination to occur with a status code specified by *status*.

The **_exit** function is always called implicitly at the end of an **exit( )** function call.

Notes

A call to **_exit** will never return.

For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

fileno — Gets File Descriptor

Syntax

```
#include <mriext.h>
char fileno (FILE * stream);
```

Class

Microtec C Extension (implemented as a macro)

Description

The **fileno** macro returns the file descriptor associated with *stream*.

Notes

The macro will yield undefined results if *stream* does not point to an open file.

ftoa — Converts Floating-Point Number to ASCII String

Syntax

```
#include <mriext.h>
int ftoa (double value, char *str, int format, int prec);
```

Class

Microtec C Extension

Description

The **ftoa** function converts the floating-point number *value* to a **NULL**-terminated ASCII string with a specified format and precision. The output is placed in the area pointed to by *str*. No leading blanks or zeros are produced.

The conversion format character *format* (which must be 'f', 'e', 'E', 'g', or 'G') and the precision *prec* are the same as those defined for the **printf** function. Note that the parameter *prec* is the same as the *digits* parameter in **printf** and that even though *format* is declared as an `int`, it can be specified as a character (as shown earlier). The **printf** function calls **ftoa** to convert floating-point numbers.

The **ftoa** function returns the number of characters placed in the output string, excluding the **NULL** terminator.

Notes

If an invalid floating-point number is passed to the **ftoa** function, the behavior is unpredictable.

Example

```
int len;
char num[15];

len = ftoa(2.15, num, 'g', 3);
```

getl — Reads a Long Integer From a Stream

Syntax

```
#include <mriext.h>
long getl (FILE * stream);
```

Class

Microtec C Extension (implemented as a function)

Description

The function **getl** reads a `long` integer from *stream* and returns it. **getl** returns **EOF** if the **End-Of-File** is read. Any `long` integer read from the *stream* must have been written with a **putl** function.

Notes

getl will yield unpredictable results if *stream* does not point to an open file.

This function can modify the static array **_iob**; no other static data is modified.

getw — Reads a Short Integer From a Stream

Syntax

```
#include <mriext.h>
int getw (FILE *stream);
```

Class

Microtec C Extension

Description

The function **getw** reads a **short int** from *stream* and returns it as an `int`. **getw** returns **EOF** if the **End-Of-File** is read.

Any short integer read from *stream* must have been written with a **putw** function.

Notes

The function **getw** will yield unpredictable results if *stream* does not point to an open file.

The function **getw** can modify the static array **_iob**; no other static data is modified.

isascii — Tests for an ASCII Character

Syntax

```
#include <mriext.h>
int isascii (int c);
```

Class

Microtec C Extension (implemented as a function and as a macro)

Description

The **isascii** function or macro returns a nonzero number if the character in *c* is an ASCII character. It returns 0 otherwise. ASCII characters range in value from 0 to 127.

The macro is invoked by default. If you want to call the function, you must do one of the following:

- Enclose the name in parentheses:

  ```
  (isascii)(c);
  ```

- Undefine the **isascii** macro with **#undef** before using it:

  ```
  #undef isascii
  ```

Notes

None.

itoa — Converts Integer to ASCII String

Syntax

```
#include <mriext.h>
int itoa (int i, char *cp);
```

Class

Microtec C Extension

Description

The **itoa** function converts the signed integer *i* to a **NULL**-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The **itoa** function returns the number of characters placed in the output string, excluding the **NULL** terminator.

Notes

None.

itostr — Converts Unsigned Integer to ASCII String

Syntax

```
#include <mriext.h>
int itostr (unsigned u, char *cp, int base);
```

Class

Microtec C Extension

Description

The **itostr** function converts the `unsigned` integer *u* to a **NULL**-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The **itostr** function returns the number of characters placed in the output string, excluding the **NULL** terminator.

Notes

None.

lseek — Sets the Current Location in a File

Syntax

```
long lseek (int fd, long offset, int whence);
```

Class

System Function

Description

The current position in the file is updated to the value given in *offset*. The *whence* argument indicates how to interpret the offset:

whence = 0	*offset* is relative to the beginning of the file.
whence = 1	*offset* is relative to the current position.
whence = 2	*offset* is relative to the end of the file.

The **lseek** function returns the resulting file position as measured in bytes from the beginning of the file. If a failure occurs, -1 is returned.

Notes

The function **lseek** is implemented simply as a stub that returns 0. Using this function can generate a link-time error message indicating that the **_WARNING_lseek_stub_used** symbol is unresolved. If your program calls the **lseek** stub, it will write the message:

```
WARNING - lseek (stub routine) called
```

to **stderr**. You must provide your own **lseek** function to allow it to work in your environment. For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

The function **lseek** can modify the static array **_iob**; no other static data can be modified.

ltoa — Converts Long Integer to ASCII String

Syntax

```
#include <mriext.h>
int ltoa (long value, char *cp);
```

Class

Microtec C Extension

Description

The **ltoa** function converts the `long` signed integer *value* to a **NULL**-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The **ltoa** function returns the number of characters placed in the output string, excluding the **NULL** terminator.

Notes

None.

ltostr — Converts Unsigned Long Integer to ASCII String

Syntax

```
#include <mriext.h>
int ltostr (unsigned long ul, char *cp, int base);
```

Class

Microtec C Extension

Description

The **ltostr** function converts the `unsigned long` integer *ul* to a **NULL**-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The **ltostr** function returns the number of characters placed in the output string, excluding the **NULL** terminator.

Notes

None.

max — Returns the Greater of Two Values

Syntax

```
#include <mriext.h>
max (a, b);
```

Class

Microtec C Extension (implemented as a macro)

Description

The macro **max** returns the larger of *a* and *b*. *a* and *b* can be any valid C expression yielding an integral, floating-point, or pointer type. **max** performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Either *a* or *b* can be evaluated more than once.

memccpy — Copies Characters in Memory

Syntax

```
#include <mriext.h>
char *memccpy (char *s1, const char *s2, int c, size_t i);
```

Class

Microtec C Extension

Description

The **memccpy** function copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied or after *i* characters have been copied, whichever comes first. This function returns a pointer to the character after the copy of *c* in *s1*.

The **memccpy** function returns a null pointer if *c* was not found in the first *i* characters of *s2*.

Notes

If ANSI features are disabled, the parameter *i* is type **unsigned int** instead of `size_t`. See section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for information on the compiler command line option that disables ANSI features.

memclr — Clears Memory Bytes

Syntax

```
#include <mriext.h>
char *memclr (char *cp, size_t i);
```

Class

Microtec C Extension

Description

The **memclr** function sets the first *i* bytes in memory area *cp* to zero. The **memclr** function returns *cp*.

Notes

If ANSI features are disabled, the parameter *i* is type **unsigned int** instead of `size_t`. See the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for information on the compiler command line option that disables ANSI features.

min — Returns the Lesser of Two Values

Syntax

```
#include <mriext.h>
min (a, b);
```

Class

Microtec C Extension (implemented as a macro)

Description

The macro **min** returns the lesser of *a* and *b*. *a* and *b* can be any valid C expression. *a* and *b* can be any integral, floating-point, or pointer type. **min** performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Either *a* or *b* can be evaluated more than once.

open — Opens a Specified File

Syntax

```
int open (char *filename, int mode);
```

Class

System Function

Description

The **open** function opens an existing file for reading, writing, or updating. The **NULL**-terminated string *filename* must correspond to a valid file name on the target operating system.

The function opens the file named by *filename* in the mode indicated by *mode*. The values for *mode* and the corresponding symbols are shown in Table 5-5.

Table 5-5. Mode Values for the open Function

Mode Value	Symbol	Description
0x0000	O_RDONLY	Opens for read only.
0x0001	O_WRONLY	Opens for write only.
0x0002	O_RDWR	Opens for read and write.
0x0008	O_APPEND	Performs writes at **EOF**.
0x0200	O_CREAT	Creates a file.
0x0400	O_TRUNC	Truncates the file.
0x4000	O_FORM	Is a text file.
0x8000	O_BINARY	Is a binary file.

Notes

The function **open** is implemented simply as a stub that returns 0. Using this function can generate a link-time error message indicating that the **_WARNING_open_stub_used** symbol is unresolved. If your program calls the **open** stub, it will write the message:

```
WARNING - open (stub routine) called
```

to **stderr**. You must provide your own **open** function to allow it to work in your environment. See Chapter 9, *Embedded Environments*, in this manual for more information.

The function **open** can modify the static array **_iob**; no other static data is modified.

The **open** function should return -1 if the file does not exist, if the file cannot be read or written, or if too many files are open.

putl — Writes a Long Integer to a Stream

Syntax

```
#include <mriext.h>
long putl (long l, FILE *stream);
```

Class

Microtec C Extension

Description

The function **putl** writes a `long` integer to *stream*. **putl** returns *l*.

Notes

putl will yield unpredictable results if *stream* does not point to an open file.

This function can modify the static array **_iob**; no other static data is modified.

putw — Writes a Short Integer to a Stream

Syntax

```
#include <mriext.h>
int putw (int w, FILE *stream);
```

Class

Microtec C Extension

Description

The function **putw** writes a short integer to *stream*. **putw** returns *w*.

Notes

putw will yield unpredictable results if *stream* does not point to an open file.

This function can modify the static array **_iob**; no other static data is modified.

read — Reads Bytes From a Specified File

Syntax

```
int read (int fd, char *buffer, int nbyte);
```

Class

System Function

Description

The **read** function reads *nbyte* bytes from the file associated with *fd* into the buffer pointed to by *buffer*. The file descriptor *fd* is returned by **open** or **creat**.

The **read** function returns the number of bytes actually read or 0 when **End-Of-File** is reached. On error, the **read** function returns -1.

Notes

For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

sbrk — Allocates Memory Space

Syntax

```
void *sbrk (int size);
```

Class

System Function

Description

The **sbrk** function allocates additional memory from the system. A new block of memory, at least *size* bytes, is allocated and a pointer to the start of this memory block is returned.

The **sbrk** function returns a pointer to the starting address of the memory block or -1 if the requested amount of memory cannot be allocated.

Notes

For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

swab — Swaps Odd and Even Bytes in Memory

Syntax

```
#include <mriext.h>
void swab (char *source, char *dest, int count);
```

Class

Microtec C Extension

Description

The **swab** function moves data from *source* to *dest*, swapping odd and even bytes in the process. The parameter *count* should be even. If *count* is odd, the last byte of *source* is not moved.

Notes

If *source* and *dest* memory spaces overlap, the behavior is undefined.

toascii — Converts a Byte to ASCII Format

Syntax

```
#include <mriext.h>
int toascii (int c);
```

Class

Microtec C Extension (implemented as a function and as a macro)

Description

The **toascii** function converts the value of its argument *c* to an ASCII character by truncating all but the lower-order 7 bits. The argument *c* is not modified. **toascii** returns a value in the range 0 to 127.

The macro is invoked by default. If you want to call the function, you must do one of the following:

- Enclose the name in parentheses:

  ```
  (toascii)(c);
  ```

- Undefine the **toascii** macro with **#undef** before using it:

  ```
  #undef toascii
  ```

Notes

None.

_tolower — Converts Characters to Lowercase Without Argument Checking

Syntax

```
#include <mriext.h>
int _tolower (int c);
```

Class

Microtec C Extension (implemented as a function and as a macro)

Description

The **_tolower** function converts the value of its argument *c* to a lower-case letter. The conversion is performed whether or not *c* is an upper-case letter. The argument *c* is not modified.

The macro is invoked by default. If you want to call the function, you must do one of the following:

- Enclose the name in parentheses:

  ```
  (_tolower)(c);
  ```

- Undefine the **_tolower** macro with **#undef** before using it:

  ```
  #undef _tolower
  ```

Notes

If *c* is not an upper-case letter, the result is undefined.

_toupper — Converts Characters to Uppercase Without Argument Checking

Syntax

```
#include <mriext.h>
int _toupper (int c);
```

Class

Microtec C Extension (implemented as a function and as a macro)

Description

The **_toupper** function converts the value of its argument *c* to an upper-case letter. The conversion is performed whether or not *c* is a lower-case letter. The argument *c* is not modified.

The macro is invoked by default. If you want to call the function, you must do one of the following:

- Enclose the name in parentheses:

  ```
  (_toupper)(c);
  ```

- Undefine the **_toupper** macro with **#undef** before using it:

  ```
  #undef _toupper
  ```

Notes

If *c* is not a lower-case letter, the result is undefined.

unlink — Unlinks a File Name

Syntax

```
int unlink (char *filename);
```

Class

System Function

Description

The **unlink** function removes the ability to access the file named by *filename*. Generally, this is accomplished by removing the directory entry for *filename*. If no other links or directory entries for the file exist, then the file is deleted.

Notes

The function **unlink** is implemented simply as a stub that returns 0. Using this function can generate a link-time error message indicating that the **_WARNING_unlink_stub_used** symbol is unresolved. If your program calls the **unlink** stub, it will write the message:

```
WARNING - unlink (stub routine) called
```

to **stderr**. You must provide your own **unlink** function to allow it to work in your environment. See Chapter 9, *Embedded Environments*, in this manual for more information.

The **unlink** function should return -1 if the operation fails.

write — Writes Bytes to a Specified File

Syntax

```
int write (int fd, char *buffer, int nbyte);
```

Class

System Function

Description

The **write** function writes *nbyte* bytes from a buffer to the file associated with the file descriptor *fd*. The starting buffer address is specified by *buffer*. The file descriptor *fd* is returned by **open** or **creat**. Up to 64K bytes may be written, starting at the current file position.

The **write** function returns the number of bytes actually written or a -1 if an error occurs.

Notes

For more information on system functions, see Chapter 9, *Embedded Environments*, in this manual.

zalloc — Allocates Data Space Dynamically

Syntax

```
#include <mriext.h>
void *zalloc (size_t size);
```

Class

Microtec C Extension

Description

The **zalloc** function allocates a new area of memory for program use and returns a pointer to that area. The size of the area allocated is equal to *size* bytes. The **zalloc** function sets the area to zeros. The space allocated is guaranteed to start on a boundary that is suitable for aligning any type.

If the requested amount of space cannot be allocated, **zalloc** returns a null pointer. Allocations are provided from a data structure called the heap, which is located in the stack section.

Notes

The function **zalloc** can modify the static structure **_membase** or the static variables **_avail** or **_badlist**; no other static data is modified.

If ANSI features are disabled, *size* is type **unsigned int** instead of `size_t`. See the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for information on the compiler command line option that disables ANSI features.

Internal Data Representation 6

Overview

This chapter describes internal data formats of the C language and the run-time organization of C programs for the 68000 family of microprocessor-based systems.

Storage Layout

Memory is accessed according to the type of processor you have. There are two basic types of processors:

- Big endian: addresses objects in memory from the "big" or most significant byte of a data object at a base memory address and stores less significant bytes at successively higher memory addresses. The preprocessor symbol **_BIG_ENDIAN** is defined for these processors.
- Little endian: addresses objects in memory from the "little" or least significant byte of a data object at a base memory address and stores more significant bytes at successively higher memory addresses. The preprocessor symbol **_LITTLE_ENDIAN** is defined for these processors.

The 68000 family consists of big endian processors.

When addressing the memory shown in Figure 6-1, the instruction to load address *N* is handled in two different ways. A big endian processor will consider *N* the more significant byte with *N*+1 as the less significant byte. A little endian processor considers *N*+1 the more significant byte with *N* as the less significant byte (see Figure 6-2).

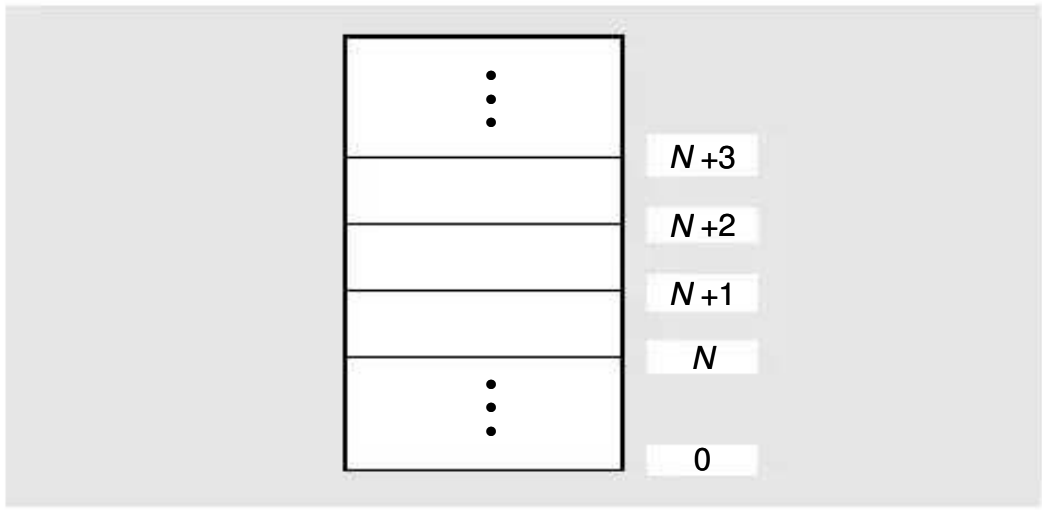

Figure 6-1. Memory Layout

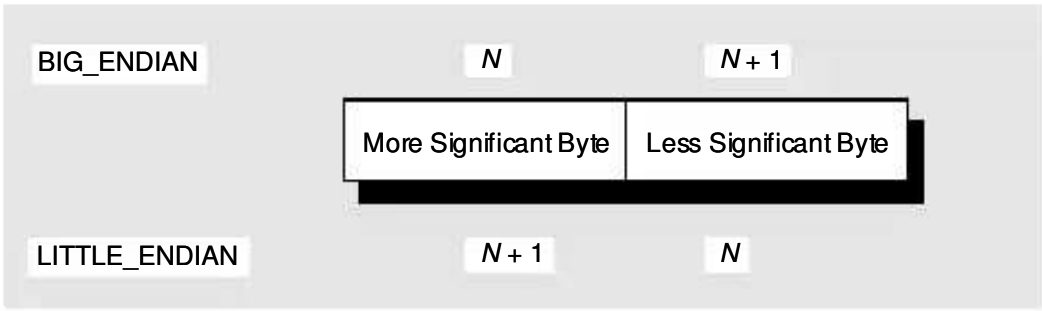

Figure 6-2. Loading Depending on Processor Type

For a 4-byte quantity with address *N* in a big endian processor, *N* represents the address of the most significant byte of the high-order word; the low-order word is located at address *N*+2, leaving the least significant byte at address *N*+3. The little endian processor treats *N*+3 as the most significant byte with *N* as the least significant byte. Figure 6-3 shows a 4-byte quantity.

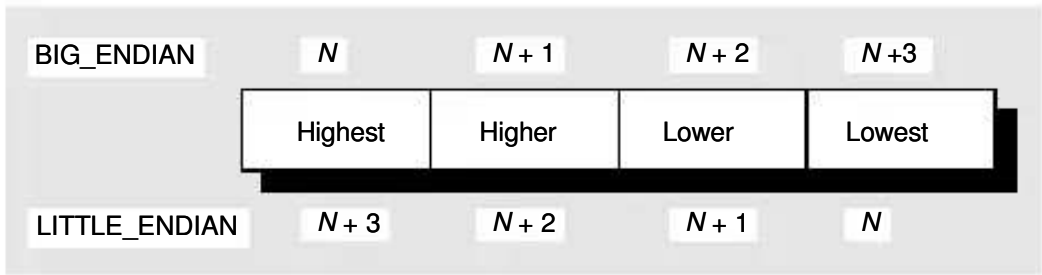

Figure 6-3. Byte Ordering in Words

Note

The term “word” refers to 2 bytes while “long word” refers to 4 bytes.

Target Parameters

Certain elements of the ANSI C language such as strings and identifiers are controlled by parameters based on your target microprocessor. The MCC68K Compiler uses the parameters set by the 68000 family of microprocessors. Table 6-1 lists the limits on these parameters.

Table 6-1. Parameter Limits

Parameter	Limit
Maximum number of characters in a string	2048
Number of initial unique characters in an identifier	512

Data Type Summary

Data types for the 68000 family include both scalar and complex data types.

Table 6-2 shows the size and value range of the scalar data types. Note that floating point value ranges are approximate.

The range of values is in decimal representations, and the alignment is in bytes.

Table 6-2. Scalar Data Types

Data Type	Size	Range	Alignment (Odd-Address Restricted Processors)	Alignment (Odd-Address Unrestricted Processors)
signed char	8 bits = 1 byte	-128 to 127	1	1
unsigned char	8 bits = 1 byte	0 to 255	1	1
short int	16 bits = 2 bytes	-32768 to 32767	2	2
unsigned short int	16 bits = 2 bytes	0 to 65535	2	2
enum	32 bits = 4 bytes		2	4
int	32 bits = 4 bytes	-2147483648 to 2147483647	2	4
unsigned int	32 bits = 4 bytes	0 to 4294967295	2	4
pointer	32 bits = 4 bytes		2	4
long int	32 bits = 4 bytes	-2147483648 to 2147483647	2	4
unsigned long int	32 bits = 4 bytes	0 to 4294967295	2	4
float	32 bits = 4 bytes	$1.18 * 10^{-38}$ to $3.4 * 10^{38}$	2	4
double	64 bits = 8 bytes	$1.18 * 10^{-308}$ to $3.4 * 10^{308}$	2	4
long double	64 bits = 8 bytes	$1.18 * 10^{-308}$ to $3.4 * 10^{308}$	2	4

Note

If the keyword **unsigned** does not appear, the **char** data type will be considered to be signed unless the **-Ku** option is used. In this case, it will be considered to be unsigned.

Table 6-3 shows the size and value range of the complex (aggregate) data types.

Table 6-3. Complex (Aggregate) Types

Complex Types	Size
array	Combined size of elements
struct	Combined size of members (plus possible padding)
union	Size of largest member

Pointers

Pointers occupy four bytes and are aligned as shown in Table 6-2.

Type Conversion

The compiler performs data type conversions under the following circumstances:

- When arguments are passed to a function that does not have a prototype. In this case, the function's arguments are integrally promoted as described in the section *Mixed Operands* in this chapter.
- When the data type of an operand is forced to be converted by type casting.
- When a binary operator is used (commonly referred to as "usual arithmetic conversions").
- When two or more operands of different types appear in an assignment expression. The right operand is converted to the type of the left operand. The conversion of mixed operands is described in the section *Mixed Operands* in this chapter.

Mixed Operands

Mixed operands of binary operators are converted according to an order of precedence, as shown in Table 6-4.

Table 6-4. Mixed Operand Conversion

Operand Type	Compiler Action
Either is **long double**	Converts other to **long double**.
Either is **double**	Converts other to **double**.
Either is **float**	Converts other to **float**.
Either is **unsigned long**	Converts other to **unsigned long**.
One is **long** and other is **unsigned**	Converts both to **unsigned long**.
Either is **long**	Converts other to **long**.
Either is **unsigned**	Converts other to **unsigned**.

If the operands do not fit into the preceding type categories, the compiler performs the integral promotions shown in Table 6-5 to ensure that both operands have type **int**.

Table 6-5. IntegralPromotion

Operand Type	Compiler Action
char	Sign-extends operand to **int** unless specified otherwise when the compiler is invoked [a]
signed char	Sign-extends operand to **int**
unsigned char	Zero-extends operand to **int**
short int	Sign-extends operand to **int**
long int	Operand treated same as **int**
unsigned short int	Zero-extends operand to **int**

a. See the section *Produce Minor Code Generation Variations* in Chapter 2, *UNIX/DOS User's Guide*, for more information.

Type Casting

Type casting forces the conversion of an expression to the specified data type. Casting an integral type to a larger type causes sign-extension or zero-padding according to the type of the value being cast.

When casting an integral type to a smaller integral type, the low-order bits are retained.

When casting a floating-point type to another floating-point type, the high-order bits of the fractional part are retained.

When casting an integral type to a floating-point type (or vice versa), the bits change completely.

Examples:

```
(char *) 5              /* Treat 5 as a pointer to a char */
(unsigned) -1           /* Value: 4294967295 */
(long) -128             /* Value: -128, 4-byte */
                        /* representation */
(unsigned long) -128    /* Value: 4294967168 */
(char) 4294967167       /* Value: 127 */
(int) 65535             /* Value: 65535 */
p2 = (type2 *) p1;      /* Assign p1's value to p2, */
                        /* even though p2 points to a */
                        /* different type */
```

Function Operations

All executable statements in a C program are contained in functions. A function is called from an expression (possibly with values passed as parameters). The function

performs a computation and (optionally) returns a result to the caller.
Functions can be declared, declared with a body, or used without being declared. In each case, the function call syntax is the same. The following sections describe Microtec extensions to function declarations.

Passing Prototyped Parameters Smaller Than int

A function prototype declaration allows the C compiler to verify that functions are defined and used consistently. A function prototype specifies the number and type of arguments accepted by the function. Microtec strongly recommends using the prototype style for function declarations. For example, a function that returns the absolute value of an integer parameter, `i`, would be declared with the prototype:

```
int abs(int i);
```

A function prototyped parameter smaller than **int** may be passed using the **-Zp2** and **-Zp4** options. For further information, refer to the description of the **-Zp2** and **-Zp4** options in section *Command Line Option Summary* in Chapter 2, *UNIX/DOS User's Guide*.

Function Declaration With interrupt Keyword

A function can be declared as an interrupt handler by using the **interrupt** keyword. Refer to the section *Interrupt Handlers* in Chapter 9, *Embedded Environments*, in this manual for more information.

Example:

```
interrupt void handler (void);
```

Structure Operations

Unlike the pre-ANSI definition of C, Microtec C allows structures to be assigned, passed as parameters to functions, and returned from functions. Structure values are placed on the stack. The amount of storage needed by a structure is the sum of the space used by its members plus any padding.

Structure Alignment

Structure members are aligned according to their type (see Table 6-2 for alignment of scalar types). The compiler may add padding between structure members and at the end of a structure. Structures are always padded so that the size is a multiple of the maximum alignment of the structure's members. However, if the largest member of the structure is a **char**, the structure will still be aligned on an even boundary.

- For processors with a default alignment of 2, a structure of the type shown in the following example is allocated 4 + 1 + (1 byte padding) + 4 = 10 bytes.
- For processors with a default alignment of 4, the same structure is allocated 4 + 1 + (3 byte padding) + 4 = 12 bytes.

These rules can be modified with the **-Za2** and **-Za4** options.

Example:

```
struct date             /* structure tag */
{
     int day;           /* structure member type int */
     unsigned char month;
                        /* structure member type */
                        /* unsigned char */
     int year;          /* structure member type int */
} holiday;              /* structure variable name */

main()
{    holiday.day = 25;
     holiday.month = 12;
     holiday.year = 1988;
}
```

In this example, the structure variable is `holiday` and the members of the structure are `day`, `month`, and `year`. The `date` is an identifier called a structure tag. This structure tag can be used for later definitions and declarations without repeating the structure members. For example:

```
struct date workday;
```

For more information on **packed** structures, refer to the section *Packed Structures* in this chapter.

Determining Structure Size

You can easily manipulate the compiler into revealing the size of a structure and the offsets of its members. Knowing the size of a type is useful for double-checking type defaults and for establishing the size of complex structures and unions.

Example:

```
struct x {
     char a;
     long b;
     char c;
};
packed struct y {
     char a;
     long b;
};

struct_size () {
     struct_size (struct x, struct y);
}
```

In this example, nonexecutable code is used to provoke a warning from the compiler that will list structure sizes. The structure types `struct x` and `struct y` are specified as parameters to the function `struct_size`. Using structure types as parameters provokes a warning message listing the structure sizes and stating that the type was replaced by the size of the structure:

```
(W) type used as argument; replaced with its sizeof: 8
```

By using the Microtec MCC68K C Compiler and the XRAY68K Debugger, you can create a program that will generate the size of the structure members.

Example:

```
#include <stddef.h>     /* include offsetof macro */


#define print_offset(tag, member) \
      printf("\noffset of %s.%s\t%d", #tag, #member, \
            offsetof (struct tag,member))

struct x {
      char a;
      long b;
};
struct y {
      char a;
      struct x b;
      long c;
      char d;
};
main () {
      print_offset (x,a);
      print_offset (x,b);
      print_offset (y,a);
      print_offset (y,b);
      print_offset (y,c);
      print_offset (y,d);
      puts("");
}
```

In this example, each structure member will be passed to the defined macro `print_offset`. The `print_offset` macro prints offsets determined by the `offsetof` macro for all structure members. To have the output directed to your terminal (UNIX only), create an XRAY include file, **myout.inc**, containing the following commands:

```
outport &_simulated_output, f="/dev/tty"
go
q y
```

Now, compile the C program and use the XRAY Debugger to generate the results:

```
mcc68k -o offset.x offset.c
xhs68k offset.x -b -i myout.inc
```

The offset of each structure member is displayed.

Bit Fields

A member of a structure can be defined as a bit field. A bit field defines the number of bits of storage that the member requires. A bit field is specified by:

basetype member_name:*number_of_bits* ;

In addition to base types **int**, **signed int**, or **unsigned int**, the Microtec **-x** compiler option provides support for bit fields of any integral type: **char**, **signed char**, **unsigned char**, **short**, **signed short**, **unsigned short**, **long**, **signed long**, **unsigned long**, **packed** enumerated types, or **unpacked** enumerated types. Storage is allocated according to this base type. The minimum size of any bit field or group of bit fields of the same base type is the number of bits in the base type. The maximum size of a single bit field is also the number of bits in its base type.

Example:

```
struct status {
     unsigned control:3;
     int data:13
} pvar;
```

Bit fields are stored in bytes starting at higher order bits. If a successive field element does not fit into the remaining space of the current base type, a new byte, word, or long word is allocated (according to the base type). The field member is placed into it starting again from the most significant order bit.
Field members of zero length are not allocated any space but cause the next member to be aligned at the next byte, word, or long word (according to the base type of the next member). Bit fields can be initialized.

When the size of the next bit field's *basetype* differs from the size of the current *basetype*, padding is added so the next bit field is aligned to its *basetype* boundary. Padding is also added if the next element is not a bit field regardless of the type of element.

Example:

```
struct {
   char bf1  : 5;
   short bf2 : 3;/* size changed, pad to next word */
   int bf3   : 2;/* size changed, pad to next long word */
   signed int bf4 : 7;
} sl;
```

Figure 6-4 shows the bit field allocation within the byte for this example.

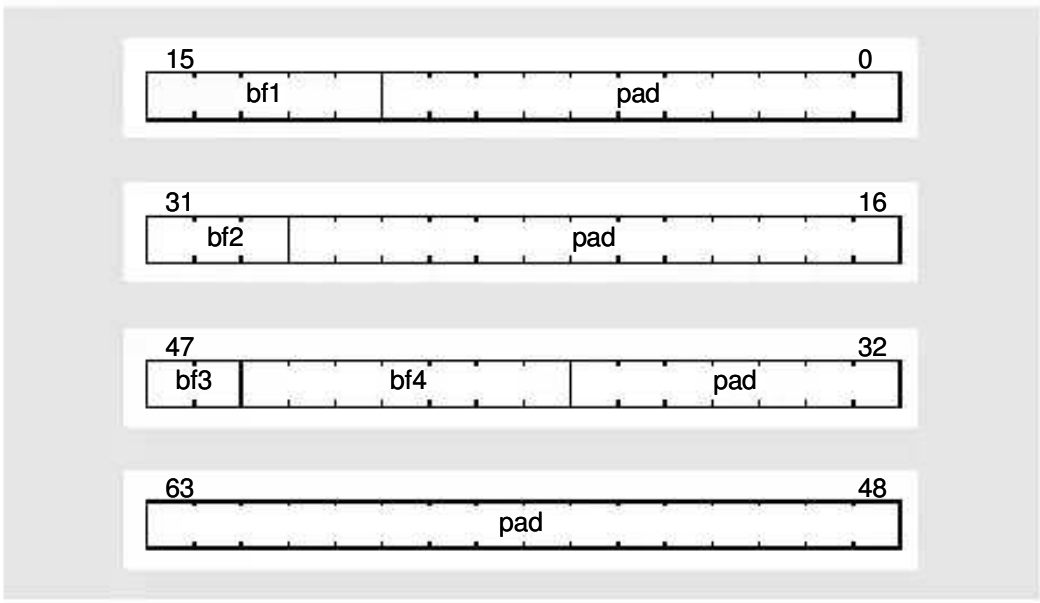

Figure 6-4. Sample Bit Field Allocation (Over 8 Bytes)

Example:

```
struct {
      char bf1 : 3;
      unsigned char bf2 : 3;  /* changed sign */
      char bf3 : 3;           /* will not fit in current
                              /* byte */
} s2;
```

Figure 6-5 shows the bit field allocation within the byte for this example.

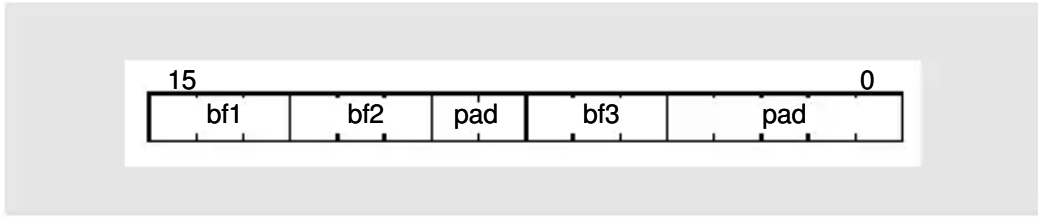

Figure 6-5. Sample Bit Field Allocation (Over 2 Bytes)

Warning

A **signed** bit field that is one-bit wide can only have the values 0 and -1 since the one bit is considered to be the sign bit.

Alignment and Packing

This section describes the alignment and packing of complex data types.

Note

This section uses the following terms to identify processors:

"Odd-address restricted" means words and long words must be accessed on even boundaries. An address error occurs when a word or long word access is made to an odd address. Processors of this type include 68000, 68008, 68010, 68EC000, 68HC000, 68HC001, 68302, and members of the CPU32 family: 68330, 68331, 68332, 68333, and 68340.

"Odd-address unrestricted" means words and long words can be accessed on odd or even boundaries without generating any errors. Processors of this type include 68020, 68030, 68040, 68060, 68EC020, 68EC030, 68EC040, 68EC060, and members of the CPU32+ family: 68349 and 68360.

Packing Bit Fields

Bit fields are packed into a long from left (most significant bit) to right (least significant bit). Each field is packed into a single long. If there is not enough room for the next bit field, it is allocated to a new long.

Bit fields cannot span 32-bit boundaries.

Alignment of Bit Fields

When dealing with data, the most important feature of an architecture is the data bus width. Fetching the size of the data bus width, usually defined as the “word” size, is the most efficient collection method (assuming information is properly aligned). C clearly defines that bit fields should not straddle a machine word unit.

Word Unit Straddling

Straddling possibilities, as shown in Figure 6-6, are:

- No straddling: the bit fields all fit into a word unit that is properly aligned.
- Simple straddling: the bit fields all fit into a word unit, but the unit is not properly aligned.
- Complex straddling: the bit fields are larger than a word unit.

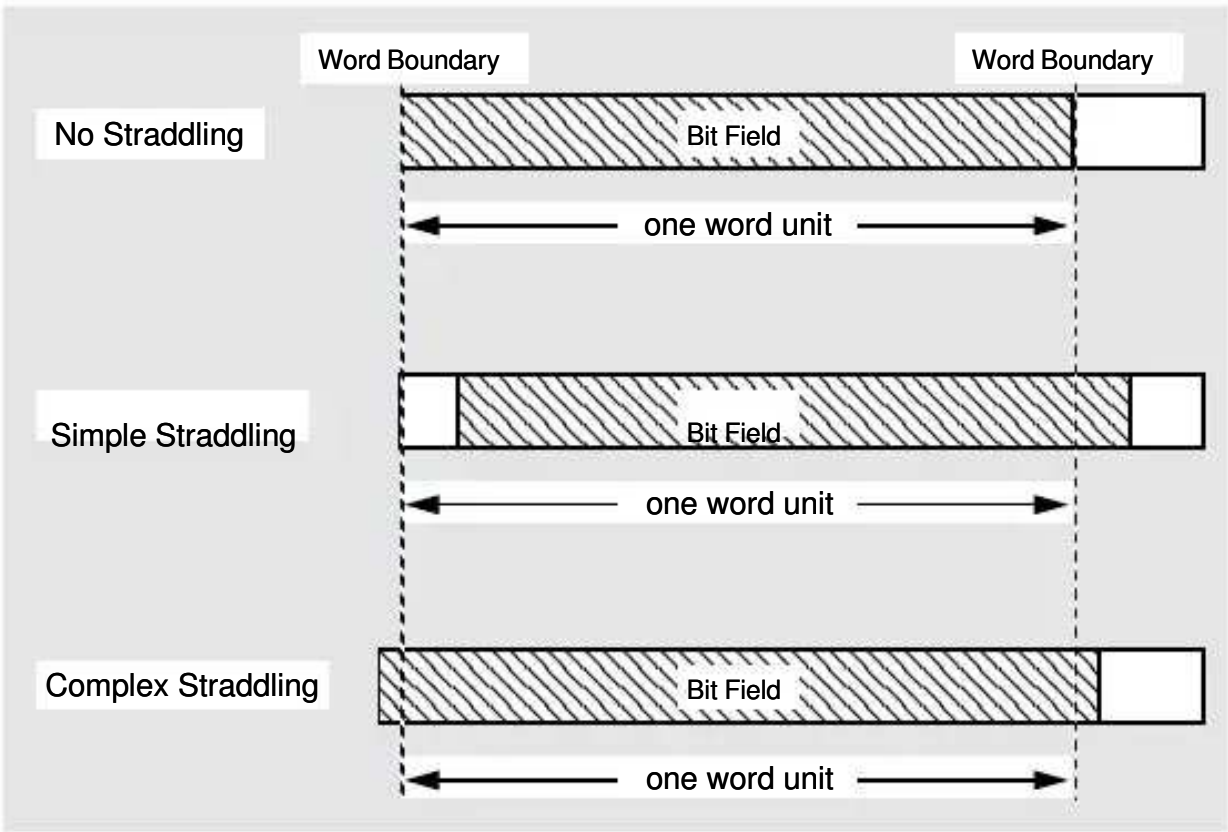

Figure 6-6. Straddling Bit Fields

If bit fields fit into a word unit, they can be easily accessed. If bit fields fit into a word unit but this unit is not aligned on the standard boundaries (simple straddling), two word fetches are needed. For cases of complex straddling, three fetches are needed.

Bit Field Alignment

A long word that contains bit fields is aligned to a 2-byte boundary for odd-address restricted processors and to a 4-byte boundary for odd-address unrestricted processors.

If the next bit field does not fit in the current word, it will be allocated in the next long.

See Figure 6-7 for bit field packing.

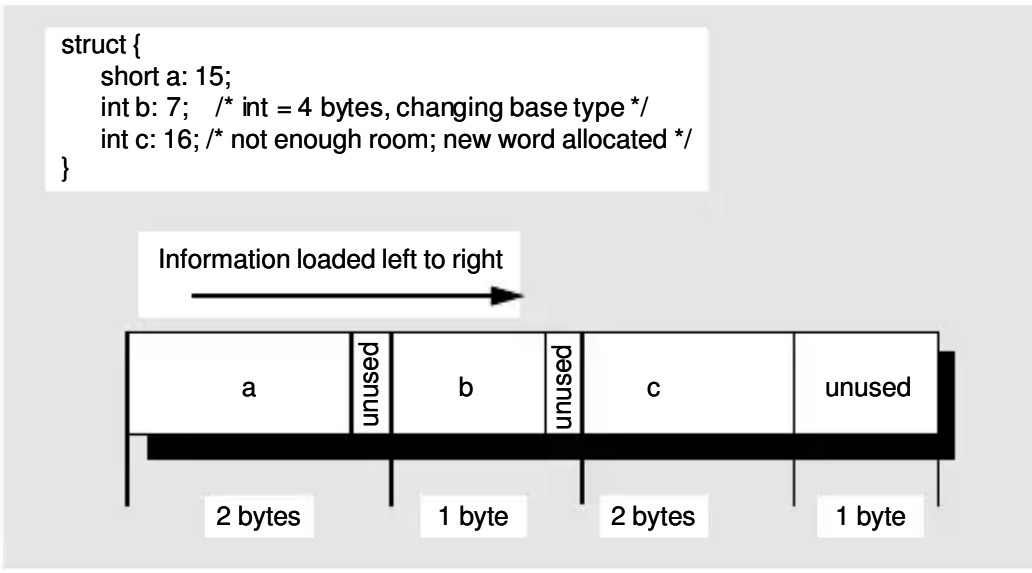

Figure 6-7. Packing Bit Fields (Big Endian)

Both signed and unsigned bit fields are implemented. Signed bit fields should be handled like signed data. Changing the sign in a sequence of bit field declarations does not introduce a new word.

See Figure 6-8 for signed bit field packing.

```
struct {
        int a: 10;
        unsigned b: 5; /* changing sign - no effect */
        int : 0; /* bit field of length 0 */
        int c: 6;
}
```

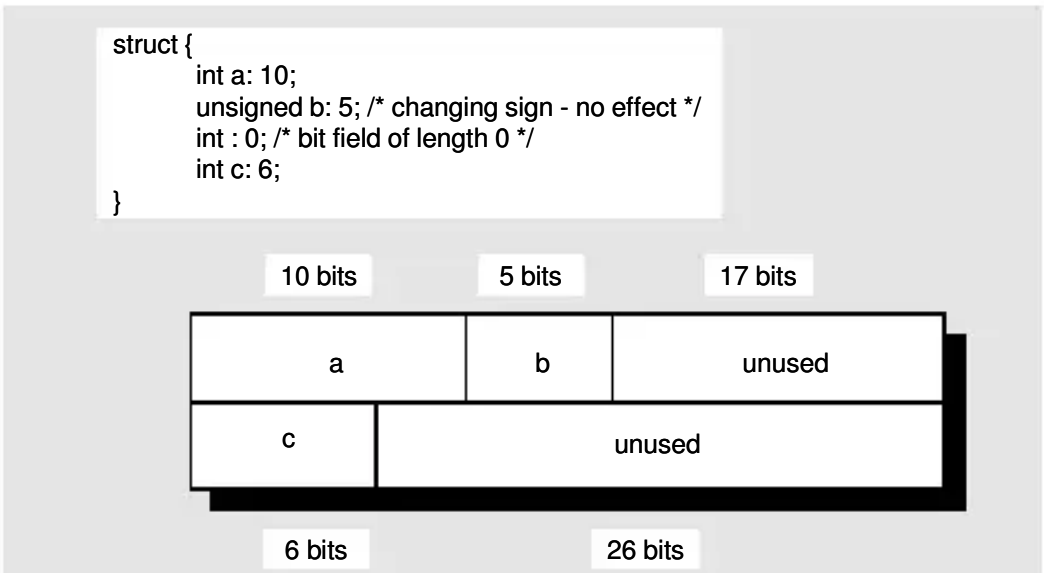

Figure 6-8. Signed Bit Fields (Big Endian)

The compiler can support bit fields of **short** (word) and **char** (byte) sizes. Specifying a bit field with length 0 forces the next bit field to be allocated to the next base type. For more information on bit field base types, refer to the section *Bit Fields* in this chapter.

Aggregates

In aggregates such as arrays, structures, and unions, each member is aligned to the boundary of its natural alignment. The aggregate itself takes the alignment of the member with the maximum alignment size.

Examples:

	Processors With a Default Alignment of 2	Processors With a Default Alignment of 4
array of double	2-byte alignment	4-byte alignment
struct with long word and byte member	2-byte alignment	4-byte alignment
union with word and byte member	2-byte alignment	2-byte alignment
array of char	1-byte alignment	1-byte alignment

Size of Aggregates

The size of an aggregate should be divisible by its alignment size. There can be unused space after the last member. This unused space should not be occupied by any effective data (see Figure 6-9).

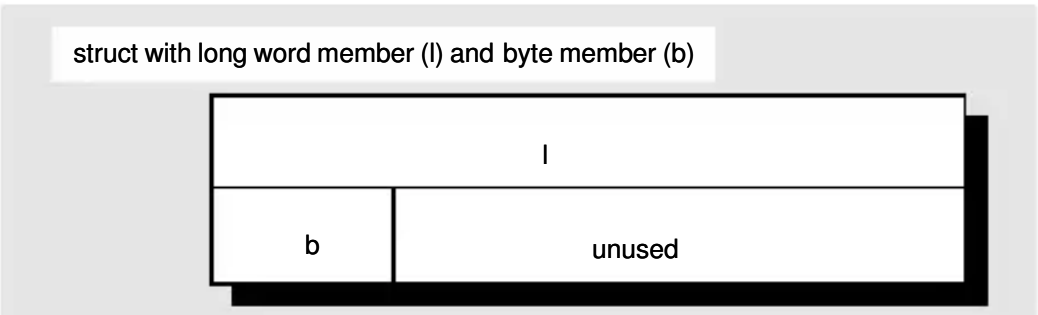

Figure 6-9. Packing of Aggregates (Big Endian)

The **sizeof** operator applied to an aggregate always returns a number divisible by its alignment size.

Examples:

```
sizeof (struct{int a; char c;})     /* size = 8 */
                                    /* 4-byte alignment */
sizeof(struct{short a,b; char c;}) /* size = 6 */
                                    /* 2-byte alignment */
sizeof(char [3][7])                 /* size = 21 */
                                    /* 1-byte alignment */
```

Packed Structures

By default, structure members will be aligned as described in the section *Aggregates* in this chapter. The **packed** keyword can be used to force no padding between structure members.

The following example shows the allocation for a packed and unpacked structure member.

Example:

```
/* PKD will be defined to be packed or unpacked */
#define PKD packed
typedef PKD struct {
     char c1;
     int i;
     char c2;
     char c3;
} str;
```

Figure 6-10 shows the unpacked and packed structures.

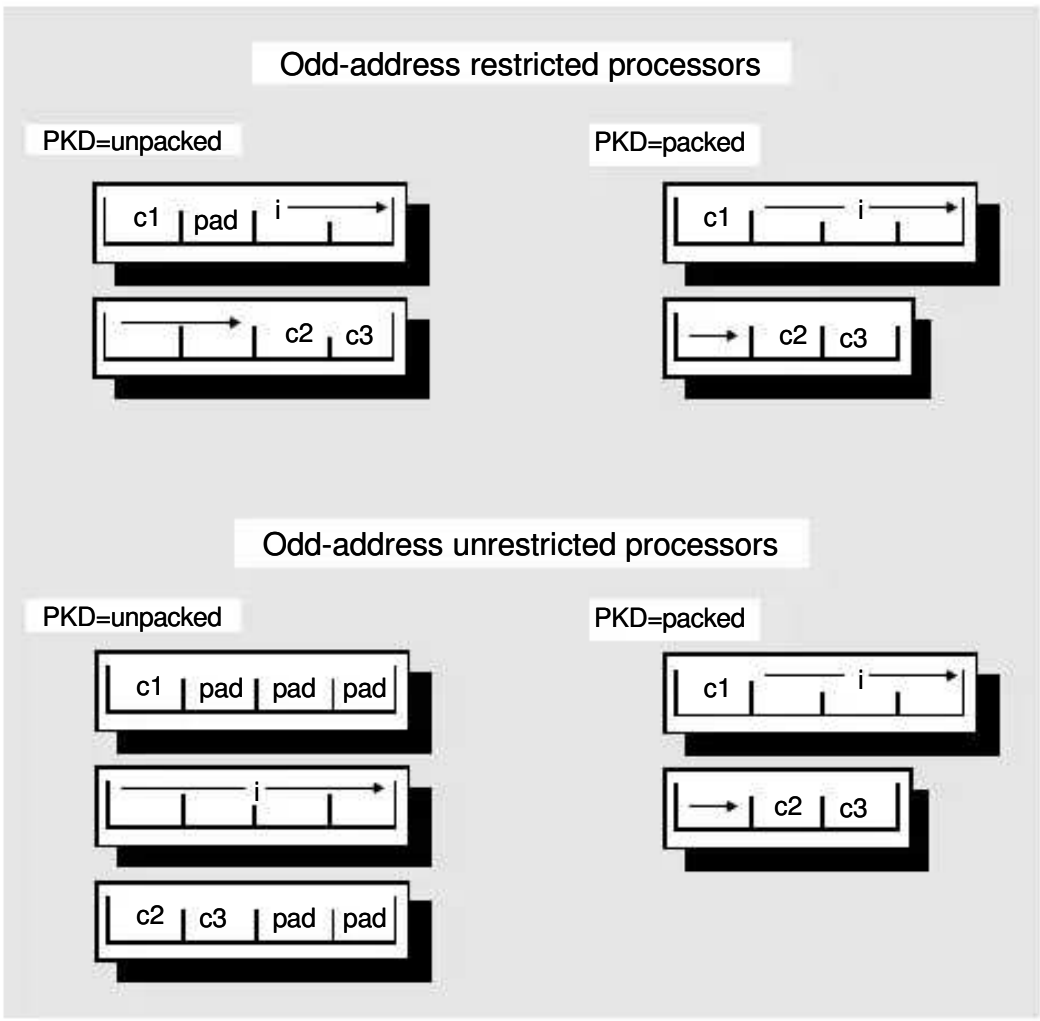

Figure 6-10. Unpacked and Packed Structures (Big Endian)

Structure Padding

Structure layout is determined by several factors:

- Bus width

  Odd-address restricted processors of the 68000 family have a 16-bit bus width. Multibyte quantities can only be directly accessed if they reside at an address that is a multiple of 2. For example, an attempt to fetch a **short** (word size) at an odd address would cause an address error. In general, this

error does not occur since variables and fields of regular structures are aligned on proper boundaries.

For packed structures, fields are not aligned. The nonaligned fields can cause word or long word fields to start at odd addresses. In these situations, fields are accessed using multiple byte and word instruction types.

Odd-address unrestricted processors of the 68000 family have 32-bit bus widths. This partition permits access to any field on any boundary (i.e., an address error never occurs). However, if the fields are not aligned to their proper boundary (words should be aligned to addresses that are multiples of 2, and long words should be aligned to addresses that are multiples of 4), the hardware uses extra cycles to access the information, which reduces performance.

Unpacked structures lay out their fields on the optimal boundary. Packed structures, however, pack elements as tightly as possible, ignoring boundaries. No special code is needed to access fields that do not start on their natural boundary since the hardware automatically takes care of the alignment (at the expense of performance).

- Field alignment requirements

  Any field other than a bit field is aligned to its natural (required) or most efficient boundary. Bit fields are aligned according to the natural boundary of the integral type specified in their definition. Table 6-6 shows the alignment requirements of C data types.

Table 6-6. Natural Boundary Alignment for the 68000 Family

Data Type	Odd-Address Restricted Processors	Odd-Address Unrestricted Processors	Notes
char	1	1	
short	2	2	
int	2	4	
long	2	4	
pointer	2	4	
float	2	4	

(cont.)

Table 6-6. Natural Boundary Alignment for the 68000 Family (cont.)

Data Type	Odd-Address Restricted Processors	Odd-Address Unrestricted Processors	Notes
double	2	4	
char : *n*	1	1	Same as char
short : *n*	2	2	Same as short
int : *n*	2	4	Same as int
long : *n*	2	4	Same as long
enum	2	4	Same as int
packed enum (1 byte)	1	1	Same as char
packed enum (2 bytes)	2	2	Same as short
packed enum (4 bytes)	2	4	Same as int
array			Same as component
union			Same as the field with the most demanding alignment requirement
struct			Same as the field with the most demanding alignment requirement
packed struct	1	1	See the **-Zm** compiler option

- Structure alignment requirements

  Structure alignment is based on the alignment of the field with the most demanding alignment requirement. A structure whose largest alignment requirement belongs to a **long** element will have the same alignment requirements as a **long**.

- Padding and trailer bytes

  The compiler can add padding bytes between two fields to force the alignment of the next field to match its natural boundary. Trailer bytes can be

added at the end of a structure to make its size a multiple of its alignment requirement. To avoid padding or trailing bytes, use packed structures.

- Size

  The size of an unpacked structure is the sum of the size of all its fields plus the size of all padding and trailer bytes.

Example:

```
struct s {
      char a;
      short b;
      int c;
      char d;
};
```

For 68000 family odd-address restricted processors, `struct s` has an alignment requirement of 2 (since the largest field, `int`, has an alignment requirement of 2) and a size of 10 (1 + 1 pad + 2 + 4 +1 + 1 trailer byte). Note that if `struct s` were packed, its size would only be 8 (1 + 2 + 4 + 1). Figure 6-11 shows the unpacked structure size.

For 68000 family odd-address unrestricted processors, `struct s` has an alignment requirement of 4 (since `int` has an alignment requirement of 4 for those processors) and a size of 12 (1 + 1 pad + 2 + 4 +1 + 3 trailer bytes). Figure 6-11 shows the structure size.

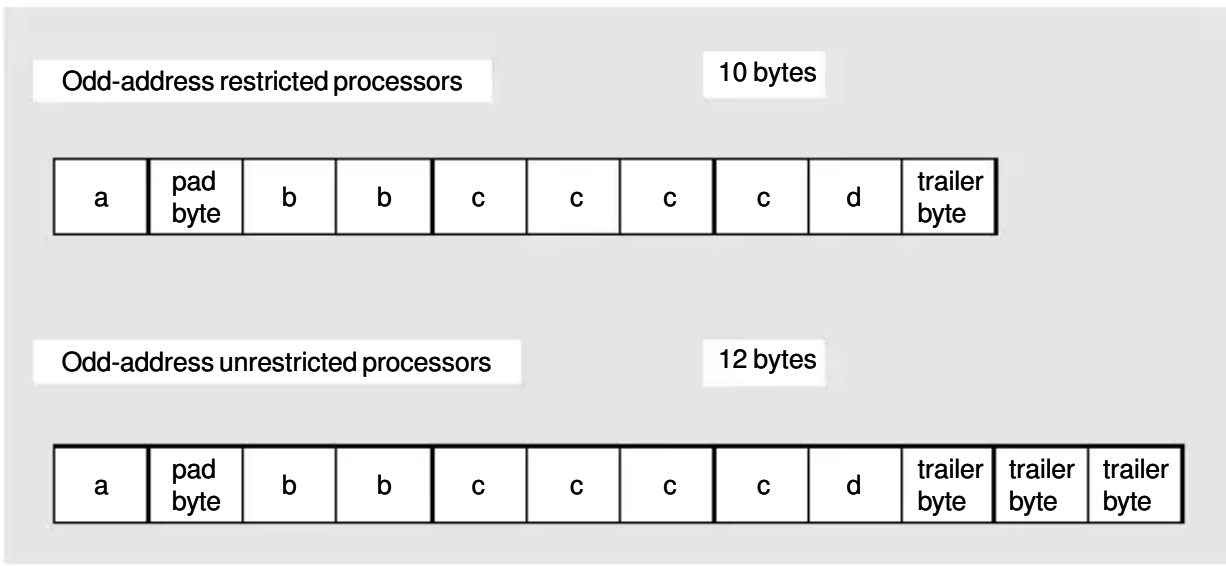

Figure 6-11. Unpacked Structure: struct s

Structure Alignment When Default Alignments Differ

When porting an application to 68000 family processors, it is important to guarantee that structure layouts are identical. Packed structures use the same alignment for all processor types. For unpacked structures, follow these guidelines:

- Use **-Za2** or **-Za4** consistently. Using **-Za2** aligns all fields of 4 bytes or greater to a multiple of 2, which can negatively affect performance on the 68020. Using **-Za4** aligns all fields of 4 bytes or greater to a multiple of 4, which maximizes efficiency on the 68020 and does not affect 68000 efficiency but may waste some space.
- Specify the options directly inside every source file using **#pragma options**:

```
#pragma options -Z4
```

Example:

```
struct {
      struct inner {
            char x;
      } a;
      long b;
} port;
```

The size of `struct port` differs according to the options specified, as shown in Table 6-7 and Figure 6-12.

Table 6-7. Effect of Compiler Options on Structure Layout

UNIX/DOS Options	Size of struct port for Odd-Address Restricted Processors	Size of struct port for Odd-Address Unrestricted Processors
Defaults: -Za2 (68000) or -Za4 (68020)	1 + 1 padding byte + 4 Total: 6 bytes	1 + 3 padding bytes + 4 Total: 8 bytes
-Za4	1 + 3 padding bytes + 4 Total: 8 bytes	1 + 3 padding bytes + 4 Total: 8 bytes

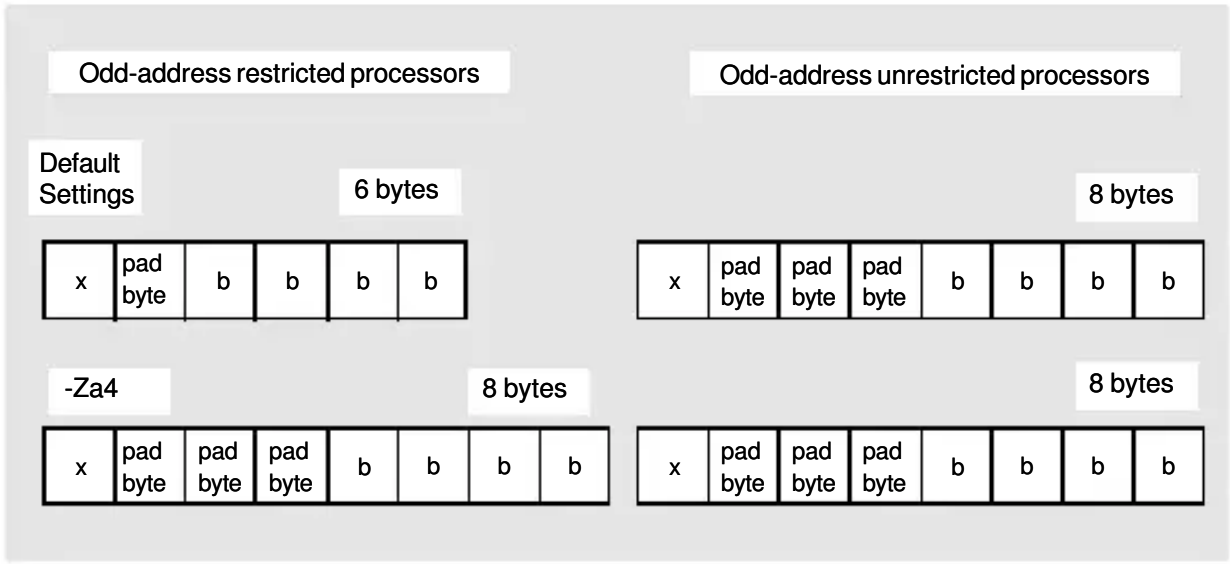

Figure 6-12. Structure Alignment Example

Warning

If all modules are not compiled with the same set of options, obscure run-time errors can occur since the "same" structure will have different layouts in different modules.

Packed Enumeration Types

A standard unpacked enumerated data type has **signed int** as an underlying type. A packed enumerated type is given instead an underlying integral type that is chosen to minimize the amount of storage necessary to efficiently hold its values.

Examples:

	Underlying Type
`enum color {red, yellow, green};`	`signed int`
`unpacked enum color {red, yellow, green};`	`signed int`
`packed enum color {red, yellow, green};`	`signed char`
`packed enum color {red=1, yellow, green=100};`	`signed char`
`packed enum color {red=127, yellow, green};`	`signed short`
`packed enum color {red=1, yellow, green=127+1};`	`signed short`
`packed enum color {red=0u, yellow, purple=255};`	`unsigned char`
`packed enum color {green=MAX_UNSIGNED_INT};`	`unsigned int`
`packed enum color {x=32768};`	`signed int`
`packed enum color {x=100000U};`	`unsigned int`

Tips About Packing

Bit fields are declared by specifying an underlying type and a bit length. For example, to describe a structure made up of 2 bits, 3 bits, 3 bits, and a **short**, the following declaration would work:

```
#pragma option -Zm1
packed struct
{
     unsigned char a:2;
     unsigned char b:3;
     unsigned char c:3;
     short s;
} three_tight_bytes;
```

If this structure is not packed, the byte total would be: 1 (`char`) + 1 byte padding (for even alignment) + 2 (`short`) = 4 bytes. As a packed structure, the byte total is 1 (`a+b+c`=8 bits) + 2 (`short`) = 3 bytes. Figure 6-13 shows the structure size when it is packed or unpacked.

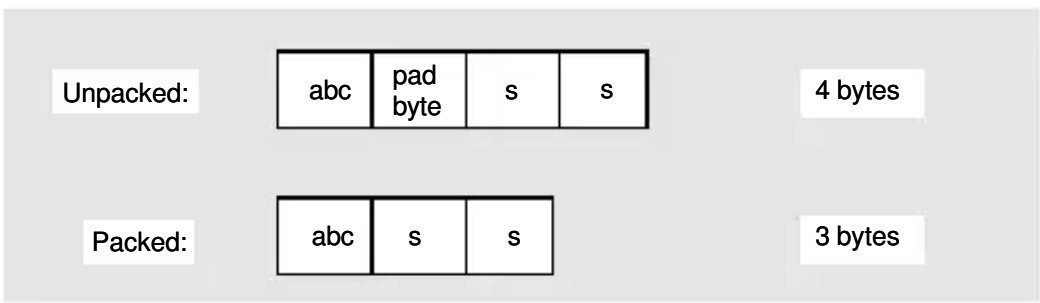

Figure 6-13. Structure three_tight_bytes

When packing a structure, a common source of confusion is that whenever the size of the underlying type changes, the remaining portion of the previous type is padded.

Example:

```
struct
{
     char a:2;
     short b:3;
     char c:3;
} four_pretty_loose_bytes;

packed struct
{
     char a:2;
     short b:3;
     char c:3;
} hope_one_tight_byte;
```

The unpacked structure contains a minimum of 4 bytes (padding might be added to the end of the structure for word alignment). The packed structure also contains 4 bytes. Since the size of a **char** (1 byte) is not the same as a **short** (2 bytes), the remaining 6 bits of `char a` are padded.

Another source of confusion is the use of enumerated data types. By definition, an enumerated type is **signed int** type.

Example:

```
typedef enum {FALSE=0, TRUE=1} boolean;

packed struct

{     unsigned char a:2;
      boolean flag1:1;
      unsigned char b:2
      boolean flag2:1;
} str;
```

By looking at this structure, you might think it will fit into one byte. However, since an enumerated type is **signed int**, it has a different size than `char`, so the bit fields are not combined. Thus, the structure size is 10 bytes: 1 (`char`) + 4 (`enum` size is **int**) + 1 (`char`) + 4 (`enum`).

For applications such as operating systems or data transmission, you may encounter data structures that have a fixed or predefined layout. If that layout is not easily accessible, the compiler provides a number of extensions to make it possible to process such structures.

For example, you cannot describe the predefined data structure in Figure 6-14 in standard C terms.

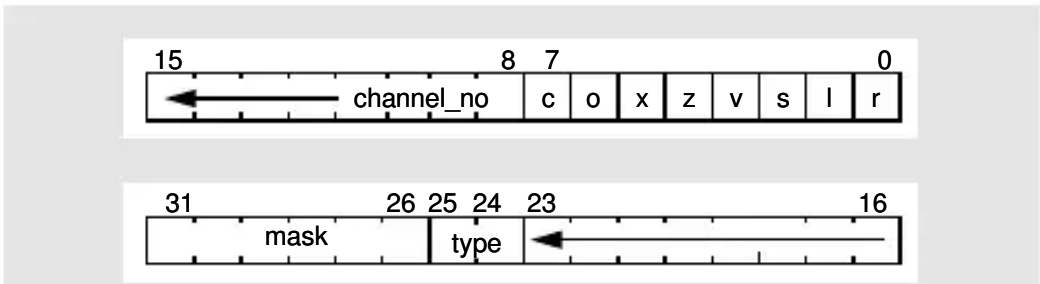

Figure 6-14. Sample Data Structure

Extensions to the Microtec compiler let you define the structure using a combination of **packed struct**, **packed enum**, and bit fields:

```
packed struct {
      packed struct {
            signed char mask:6;
            signed char type:2;
      } interrupts;
      short int channel_no;
      packed enum {c=128,o=64,x=32,z=16,v=8,
                        s=4,l=2,r=1U} flag;
} psw = { {0x1f,0}, 1, v|x|s};

assert(sizeof(psw)==4);
```

Allocation of Variables

Memory variables are allocated based on their size and frequency of use. Scalar and pointer variables generally qualify for allocation to a register unless the variable address is taken.

Register, local, and static variables are discussed in the following sections.

Register

The register storage class is used for local variables or parameters only. For each function, C allocates as many variables as possible to the hardware registers. The C language **register** storage class keyword is used like a storage class keyword, but it is not a storage class.

In general, use the **register** keyword to indicate variables that are heavily used; placing such variables in a register increases code efficiency.

Local Variables

The compiler can put local (automatic) variables into registers even if a **register** declaration is not used. Eight, sixteen, and thirty-two bit variables can be allocated into registers. The lifetime of each local variable is examined by the compiler; several variables can share the same register in one routine if possible. When a variable is allocated to a register, it will always reside in that register. However, since other variables may share the register, the value of the register may not always contain the value of the variable in question.

Variables that are declared in **register** declarations are allocated to registers before nonregister variables. This allocation lets variables be specified in their order of importance.

Static Variables

According to the C language definition, all static variables are initialized to zero automatically unless they are explicitly initialized to a value other than zero.

Static variables can be allocated to registers if they are assigned in subroutines before they are referenced. Static variables can be reorganized by the compiler, but memory might not be allocated to variables that are never used.

By default, uninitialized static variables will be allocated in the **zerovars** section. This section is cleared at program start-up time. For options that can alter this behavior, see the *Initialize Uninitialized Global Data* in Chapter 2, *UNIX/DOS User's Guide*.

Memory Allocation

Memory is byte addressed with the least significant byte at the highest address (i.e., high to low ordered). Bits are numbered with bit zero as the least significant bit.

For 68000 family odd-address restricted processors, data items that are larger than 1 byte are even-aligned. The stack and all complex data types are even-aligned. For 68000 family odd-address unrestricted processors, data items that are a multiple of 4 bytes are quad-aligned.

Data items that are a multiple of 2 bytes are even-aligned. The stack and all complex data types are quad-aligned.

Packed bit fields are allocated starting at the most significant bit, and each bit field must be fully contained in no more than four bytes.

Warning

Programs that rely on byte ordering (that is, programs that declare a variable as **int** in one module and as **char** in another) may not be portable across CPUs.

C Calling Conventions 7

Overview

This chapter describes C and assembly language calling conventions for the Microtec C compiler. This chapter provides enough information for you to write assembly language routines to interface with C language functions.

> **Note**
>
> The Microtec C compiler makes use of several intrinsic functions. These intrinsic functions are tuned for performance and do not follow any calling convention.

Parameter Passing

Assembly language routines must follow the C parameter passing conventions to pass values to, or to receive values from, C functions. C functions pass parameters by value on the stack. Parameters are always evaluated from right to left, with the first parameter being evaluated last. The next section describes the conventions on procedure calls.

Setting Parameters

All parameters are pushed on the stack and are set in the parameter area. The first parameter in the source program is placed at the lowest address as shown in Figure 7-1.

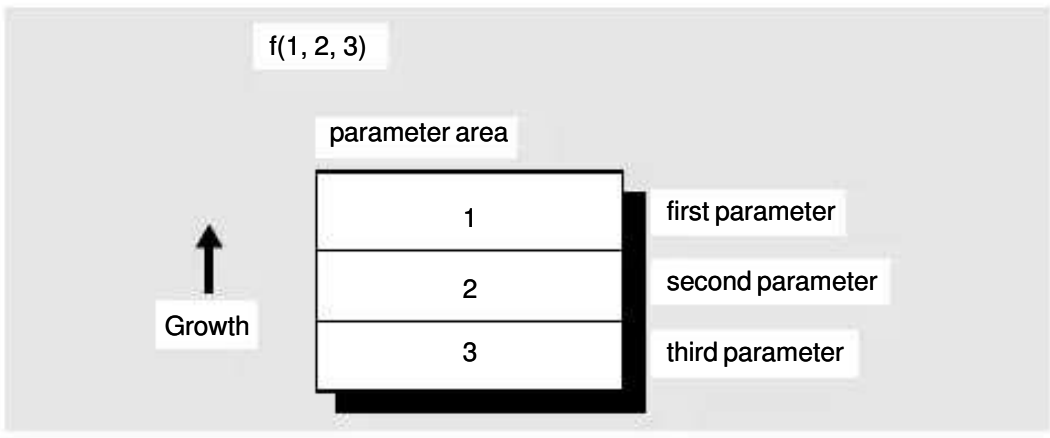

Figure 7-1. Parameter Area of the Stack

Setting Short Integers

The **-Zp2** and **-Zp4** options affect parameter promotion; see the section *Produce Minor Code Generation Variations* in Chapter 2, *UNIX/DOS User's Guide*, for more information.

Without function prototyping, **short** integers (byte or word) are pushed on the stack as long words. Normally, a **short** integer is extended by padding or sign extension before it is pushed (see Figure 7-2).

If the **-Zp4** option is specified, if a **short** argument is passed to a function with a matching **short** prototype parameter, it is not extended. It is pushed as a long word, but the upper two bytes are meaningless.

If function prototyping is used while the **-Zp2** option is in effect, only a short word is pushed onto the stack for a **short** integer argument.

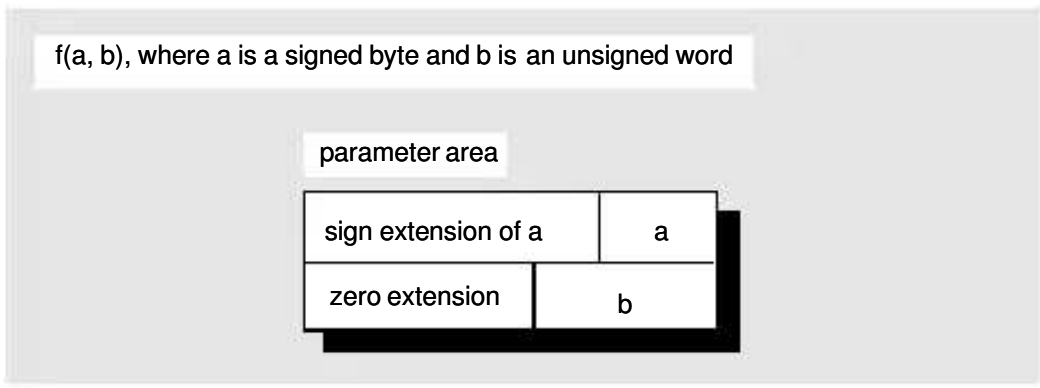

Figure 7-2. Parameter Area and Short Integers

Note that the **-Zp2** option generates incorrect code if one of the following occurs:

- A function with **char/short** parameters is defined with a prototype in one module and called from another module where it is declared without a prototype.

Example:

```
module1:

        f(char c, short s, int i)
        {
              ...
        }

module2:

        extern int f();
        main()
        {
              int i;
              char c;
              short s;

              f(c,s,i);          /* Bad code generated */
        }
```

- A function call is made by dereferencing a pointer to a function which contains the address of a function which has been declared with a prototype containing a **char/short** parameter.

Example:

```
f(char c, short s, int i)
{

}       ...

int (*fp)();
main()
{
        int i;
        char c;
        short s;

        fp = f;
        (*fp)(c,s,i);      /* Bad code generated */
}
```

Implicit Parameter for Structure/Union Return Value

When the return value type is a structure or union, the calling function pushes the pointer to the area in which the return value is set by the called function. This return value area should be allocated by the calling function as shown in Figure 7-3.

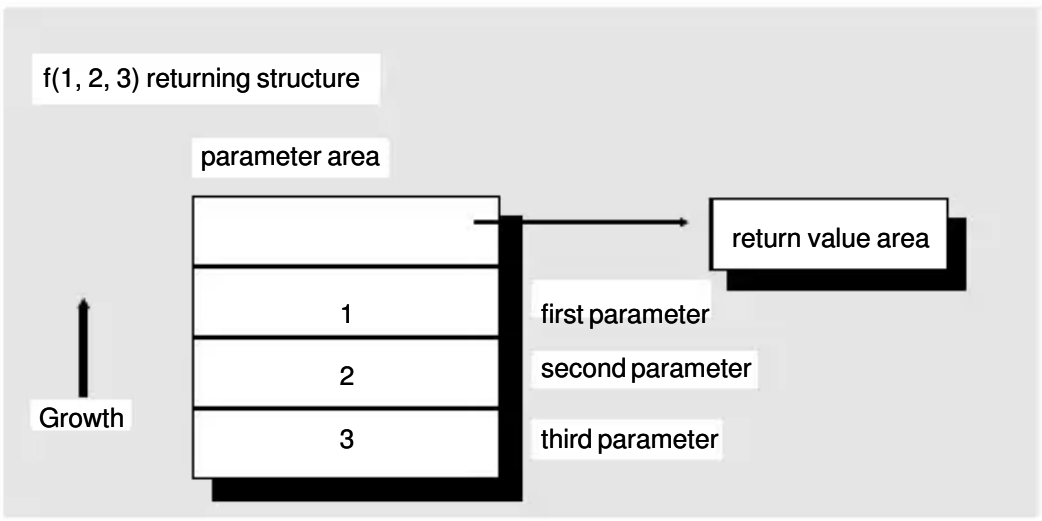

Figure 7-3. Parameter and Return Value Areas

Alignment of Structure/Union in Parameter Area

If the parameter is a structure or union, and its size is not a multiple of 2, the size of the parameter area is rounded up to be a multiple of 2 and the parameter data is set at the beginning of this area. This technique is illustrated in Figure 7-4.

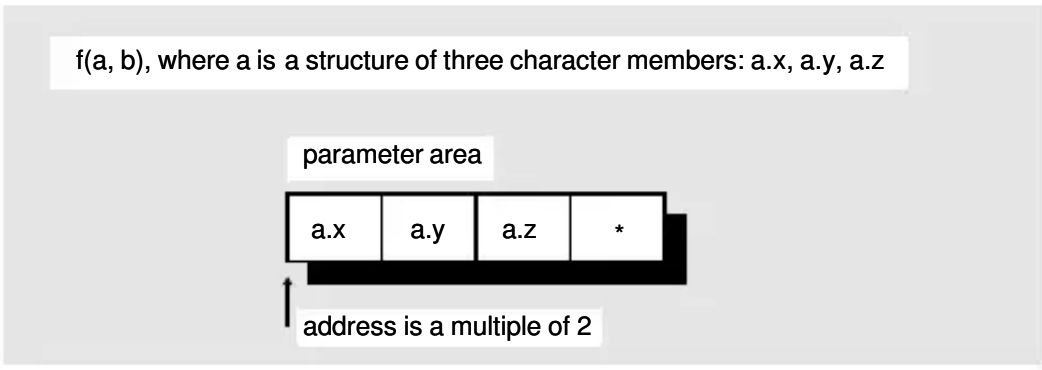

Figure 7-4. Parameter Area and Structure/Union

Function Return Values

This section describes the type of return values that are possible.

This section describes the type of return values that are possible.

Integer Return Values

An integer or pointer type return value is set in **D0**. An integer return value with a size less than 4 bytes is not extended to a long word before returning.

Floating-Point Return Values (with FPU)

A floating-point return value is set in **FP0**.

Floating-Point Return Values (without FPU)

A floating-point return value is returned in **D0/D1**, with **D0** containing the high order value.

Structure/Union Return Value

A structure or union return value is returned through the implicit first parameter which points to the return value area.

When a structure return value is assigned to another structure or pushed onto the stack as a parameter, the implicit parameter that indicates the return value area can be selected as follows:

Example:

```
struct s {
int  r, i;
} a, f();    <implicit parameter is the address of a>
:
a=f ();          PEA   _a
:     JSR    _f
:     ADDQ.L   #4,SP

<implicit parameter is the address of
 parameter area>
g(f(),1);      PEA   1
PEA   -8(A6)
JSR   _f
ADDQ.L#4,SP
LEA.L (A6),A0
MOVE.L-(A0),-(SP)
MOVE.L-(A0),-(SP)
JSR   _g
```

Popping Parameters

The parameter area is popped by the calling function.

Implicit First Parameter for Structure/Union Return Value

There is an implicit first parameter for a structure or union return value.

The code fragment below shows the caller/calling function interface.

```
int  f (int  a,...)          _f:LINKA6,#0
                        UNLK  A6
                        RTS
{
}

     :                        PEA1
     f (1);             BSR.W _f
     :                 ADDQ.L  #4,SP
```

Register and Stack Usage With Functions

The Microtec C compiler uses register **A7** as the stack pointer and register **A6** as the frame pointer. All other registers are used to store temporary variables and intermediate results.

In addition, the compiler can be forced to reserve a register (**A2** through **A6**) by using the **-Kh** option. When a register is reserved, the compiler does not generate any code that uses the reserved register. If the **A6** register is reserved, **A5** will be used as the frame pointer.

Functions that are not declared as interrupt handlers are assumed to destroy the registers **D0, D1, A0,** and **A1** (and **FP0** and **FP1** if a floating-point coprocessor is present). At the return of a function, the condition codes are undefined. All other registers that are used will be saved and restored automatically by a called C function. Assembly language routines that are called from C must also save and restore all registers that are used, except for registers **D0, D1, A0,** and **A1** (and also **FP0** and **FP1** if the floating-point coprocessor is present).

Stack Frames

Local frames for functions are automatically generated by the Microtec C compiler if a function requires local stack storage. Frames are generated by a **LINK** instruction on entry and an **UNLK** instruction on exit. When a frame is generated, local stack storage and variables are accessed on the stack with the **A6**-relative addressing mode.

> **Note**
>
> Modifying the stack pointer can cause unpredictable run-time errors.

If a function does not require local stack storage, local frames are not generated by default. When a frame is not generated, variables and parameters are accessed on the stack with the **A7**-relative addressing mode. You can force the use of local frames by using the **-Kf** option.

When a function returns, any arguments pushed on the stack are removed. If several function calls follow each other in a basic block, the stack will continue to grow until the end of the block. You can force the compiler to pop the stack after each function call by using the **-nOc** option. The following instructions shown in Table 7-1 are used depending on how many bytes need to be removed.

Table 7-1. Instructions for Removing Bytes from Stack

Number of Bytes to Remove	Instruction to be Used
1 to 8	ADDQ.L XX,SP
8 to 32768	LEA X(SP),SP
More than 32768	ADD.L X,SP

Function frames in excess of 32K bytes are considerably less efficient than smaller frames. The 68000 addressing modes are designed for 16-bit offsets. When offsets are larger than 16 bits, every machine instruction with a large offset will have one to three instructions added to it in order to bring the offset within range.

The Prologue

The prologue is the sequence of code at the beginning of a function. The prologue sets up a local stack frame and a frame pointer so that function parameters and automatic variables can be accessed through the pointer. The prologue can also include code that saves certain registers on the stack.

You can write your own assembly routines to interface to C programs. If the routines change the contents of any of the registers **D2** through **D7**, **A2** through **A5**, or **FP2** through **FP7**, you must save the old values.

The following example illustrates the recommended instruction sequence for entry to an assembly language routine.

Example:

```
entry:
        link     a6,#-N
        movem.l  reglist,-(sp)
```

In this example, *N* is the number of bytes allocated for local variables and `reglist` is the list of registers that the routine can modify.

If this sequence is used, the last argument pushed by the function call (which is also the first argument given in the call's argument list) is at address (**A6+8**). Subsequent arguments begin at (**A6** + **8** + the size of preceding arguments).

Local Variables in the Prologue

Local (automatic) variables are located on the stack with negative displacements from the frame pointer, and there can be a maximum of 2^{31}-1 bytes addressable by the frame pointer with negative displacements.

Function frames larger than 32K bytes are considerably less efficient than smaller frames. The 68000 addressing modes are designed for 16-bit offsets. When offsets are larger than 16 bits, every machine instruction with a large offset will have one to three instructions added to it to bring the offset within range. For more information, see the section *Stack Frames* in this chapter.

If you write your own applications and the assembly language routine does not have function arguments and local variables, it is not necessary to set up a local stack frame. Your routine should save the contents of the registers **D2** through **D7**, **A2** through **A5**, and **FP2** through **FP7**, if they are used.

The Epilogue

The epilogue is the sequence of code at the end of a function. The epilogue restores the saved registers and resets the stack frame to the condition at entry. It then passes control back to the calling routine with an **RTS** instruction.

Registers that are saved in the prologue should be restored first. The local stack frame can be reset by loading the stack pointer with the frame pointer. The calling routine's frame pointer is then restored by popping the saved value from the stack.

The following is the recommended instruction sequence for exiting a routine:

```
movem.l  -N(a6),reglist
unlk     a6
```

In this example, `N` is based on the number of bytes allocated for local variables and the number of registers in `reglist`.

Assembly Language Interfacing

When interfacing to an assembly language routine, it is recommended that you create a minimal local stack frame. This provides an easy way of keeping local data on the stack. It will also allow the Microtec debugger to display assembly language routine names in the procedure traceback viewport.

To write an assembly language routine that can be called from C, perform these steps:

1. Reference C global and external variables.

   Prefix all global (i.e., public) and external names with an underscore.

2. Create a local stack frame.

   Create a minimal stack frame using the **LINK** instruction. **LINK** saves the old frame pointer (**A6**) of the calling routine on the stack and makes the new frame pointer (**A6**) point to the old frame pointer.

   The **LINK** instruction can also create a local data storage area in the stack frame if needed. By creating local stack storage in this way, you will not have to pop off the exact number of arguments that were pushed onto the stack when returning from the assembly language routine.

Example:

```
link a6,#-8
```

3. Save registers.

   Save any of the registers **D2** through **D7**, **A2** through **A5**, and **FP2** through **FP7**, if they were used in the assembly language routine. If you did not create a local stack frame, you must also save the registers **A6** and **A7** if they were destroyed by the assembly language routine. All registers should be saved in numerical order immediately after the stack frame is built. Data registers should be saved first, followed by address registers, then floating-point registers.

Example:

```
movem.l   d2/d3/a2/a3,-(sp)
fmovem.x  fp2/fp3/fp4,-(sp)
```

4. Reference arguments (optional).

   You can reference arguments passed to the routine via the frame pointer (**A6**) or the stack pointer (**A7**). If you use **A7**, the offsets to the arguments must allow for the size of the local stack frame plus any saved registers. The size of the stack frame is the size of the **A6** register (4 bytes), plus the amount of local storage allocated. If you use **A6**, offsets to the arguments must allow for the size of the saved **A6** value only.

5. Return values.

   Return values should be correctly placed before returning. For more information, see the section *Function Return Values* in this chapter.

6. Restore registers.

   At the end of the assembly language routine, the saved registers and the old stack frame must be restored. This must be done immediately before the return instruction.

Example:

```
fmovem.x  -60(a6),fp2/fp3/fp4
movem.l   -24(a6),d2/d3/a2/a3
unlk      a6
rts
```

Assembly Language Routine Example

A prototype of an assembly language routine (named `_foo`) that can be called from a C function is shown as follows.

Example:

```
_foo: link a6,#0; Creates stack frame.
      ; Allocates 0 bytes of space
      ; for local data. Saves
      ; original A6. A6 now points
      ; to saved A6 value on stack.
movem.l d2/d3/a2,-(sp); Pushes 3 registers on stack.
      ; This instruction is used
      ; only if registers D2-D7,
      ; A2-A5 are used in the
      ; assembly language routine
      ; body.

      ; Stack Frame:
      ; 2nd argument       A6+12  SP+24
      ; 1st argument       A6+8   SP+20
      ; A6 -->Original A6  A6+0   SP+12
      ;       Original A2  A6-4   SP+8
      ;       Original D3  A6-8   SP+4
      ; SP -->Original D2  A6-12  SP+0
.
.
      ; Body of procedure.
.
.
movem.l -12(A6), d2/d3/a2; Gets saved registers from the
      ; stack. Values are not popped.
      ; The stack pointer is not
      ; used because other values may
      ; have been pushed onto the stack
      ; in the procedure body. This
      ; uses the A6 register, and the
      ; constant offset (12) shown
      ; above. You must compute it by
      ; adding the value given to the
      ; LINK instruction (0) and 4 *
      ; the number of address and data
      ; registers saved (3);
      ; 0+(4*3)=12. It is a negative
      ; offset because it is below the
      ; A6 register. The UNLK below
      ; will set the stack pointer back
      ; to its original value. The
      ; instruction is used only if
      ; registers D2-D7 or A2-A5 are
      ; used in the procedure body.

unlk a6; Undoes the stack frame.
      ; Restores A6 to original value.
```

```
        rts   ; Returns to caller.
```


Variable References from Assembly Routines

The compiler automatically prepends a leading underscore to all variable names. The options described in section *Modify Naming Conventions* in Chapter 2, *UNIX/DOS User's Guide*, can be used to change this behavior; however, they are not recommended because they prevent access to the standard C library functions (all library functions have a leading underscore).

Interrupt Handlers

Interrupt handlers perform the necessary functions when an interrupt occurs. They preserve the values in various registers and storage locations and transfer control to routines to service the interrupt.

By default, the Microtec C compiler assumes that the values in registers **A0**, **A1**, **D0**, and **D1** may be destroyed by each called function. Therefore, you may want to declare a function as an interrupt handler to force the values of these registers to be saved on the stack on entry to the function and to be restored on exit.

A function can be declared as an interrupt handler by using the **interrupt** keyword. Interrupt functions should have no parameters and should return the **void** type.

Example:

```
interrupt void handler (void)
{
...
}
```

By default, a function declared as an interrupt handler will return to the calling function with the **Return To Exception** (**RTE**) instruction rather than the standard **RTS** instruction.

If the **-Kr** option is used, the interrupt handler will return with an **RTS** instruction.

Run-Time Organization 8

Overview

This chapter describes the memory organization of C programs for 68000 family microprocessor-based systems.

Code Organization

The MCC68K C Compiler places each variable and function into a specific section. Table 8-1 shows the sections that are used by the compiler.

Table 8-1. Sections Generated by the Compiler

Name	Type	Contents
code	CODE	Program code
strings	CODE	String literals
literals	CODE	Compiler-generated data
const	CODE	Explicitly initialized **const** variables
ioports	DATA	Simulated I/O ports used with the XRAY Debugger
vars	DATA	Explicitly initialized non-**const** variables
zerovars	DATA	Uninitialized variables
tags	DATA	Symbol definitions to tag each function's entry and exit points
stack	DATA	Stack pointer initialized to point to the end of the section
heap	DATA	Heap pointer initialized to point to the start of the section

Each section has a name and a type. The following types are defined:

CODE	For code or data that are read-only and, therefore, can be safely placed in Read-Only Memory (ROM).
DATA	For all data that may be modified. Typically, this data is placed in Random-Access Memory (RAM).

There can be several differently named sections of the same type. For example, the **code**, **strings**, **literals**, and **const** sections are all of type **CODE**, since they are all typically read-only sections.

If a variable is not initialized in any module, it will be treated as a ‘‘C common.’’ The linker will allocate ‘‘C common’’ variables in the **zerovars** section. The compiler will allocate uninitialized static variables in the **zerovars** section. The **-X** option can be used to modify this behavior. For more information, refer to the section *Initialize Uninitialized Global Data* in Chapter 2, *UNIX/DOS User’s Guide*.

Refer to the options in section *Name the Sections* in Chapter 2, *UNIX/DOS User’s Guide*, for information on specifying alternate section names.

Changing Default Addressing

The **code** section should always be located in ROM.

Items in the **code** section are referenced according to the address mode chosen with the appropriate **-Mc** code option described in section *Generate Position-Independent Code and Data* in Chapter 2, *UNIX/DOS User’s Guide*.

Items in the **zerovars** section are referenced according to the address mode chosen with the appropriate **-Md** data option described in section *Generate Position-Independent Code and Data* in Chapter 2, *UNIX/DOS User’s Guide*. The **zerovars** section should always be located in RAM.

Refer to the appropriate **-a** option in section *Choose Address Mode for Sections* in Chapter 2, *UNIX/DOS User’s Guide* for information on specifying alternate addressing modes for the various sections.

Refer to the **-Kt** compiler option in section *Produce Minor Code Generation Variations* in Chapter 2, *UNIX/DOS User’s Guide*, for information on the **tags** section. The **tags** section should always be located in RAM. Items in the **tags** section are referenced according to the address mode chosen with the appropriate **-Md** data option (absolute, PC-relative, or **A***n*-relative) described in section *Generate Position-Independent Code and Data* in Chapter 2, *UNIX/DOS User’s Guide*.

The remaining sections (**vars**, **const**, **strings**, and **literals**) can each be located in

ROM or RAM at link time. If you are generating position-independent code or data

and you will be locating any of these sections in the nondefault area (ROM/RAM), you must tell the compiler how to address these sections.

Compiler-Generated Sections

To show how the compiler generates sections, a short example is given below. The example uses two modules: **module1.c** and **module2.c**. These modules are compiled and assembled to produce **module1.o** and **module2.o**. Finally, they are linked together to produce an executable file. The contents of **module1.c** and **module2.c** are shown in Table 8-2.

Table 8-2. Example Modules

module1.c

```
int a = 0;
int b;
static c;
const ten = 10;

main ()
{
int d;
func ("hello");
/* ... */
}
```

module2.c

```
int a;
int b;
static c;

func (char * string)
{
/* ... */
}
```

Table 8-3 shows the contents of each section contained in **module1.o** and **module2.o**. The Microtec C compiler prepends an underscore (_) to symbol names.

Table 8-3. Sections Contained in module1.o and module2.o

Name	Type	module1.o	module2.o
code	CODE	`_main`	`_func`
strings	CODE	`"hello"`	
literals	CODE		
const	CODE	`_ten`	
vars	DATA	`_a`	
zerovars	DATA	`_.S0_c`	`_.S0_c`

In **module1.o**, `_b` is an unresolved reference that has not been allocated to any section. The section for `_b` cannot be determined until linking; it may eventually reside in the **const**, **vars**, or **zerovars** sections, or even in a different user-defined section. Similarly, in **module2.o**, `_a` and `_b` are unresolved references that have not been allocated to any section. The variables `_d` and `_string` are not allocated to any section. They are accessed through the stack at run time. The two modules are linked to create an executable file. The linker combines all sections of the same name together. Table 8-4 shows the sections contained in the resulting executable file.

Table 8-4. Sections Contained in the Executable File

Name	Type	Contents
code	CODE	_main, _func
strings	CODE	"hello"
literals	CODE	
const	CODE	_ten
vars	DATA	_a
zerovars	DATA	_b, _.S0_c, _.S0_c

Notice that the two static variables named `_c` remain separate and are not linked together. Since `_b` was not present in any object file section, the linker placed it into the **zerovars** section.

The following sections describe the compiler-generated sections in more detail.

code Section (Type CODE)

The **code** section contains all program code. All code is assumed to be read-only and, therefore, can be safely placed in ROM.

strings Section (Type CODE)

The **strings** section contains the contents of all string literals. If your application does not attempt to modify string literals, then the **strings** section can be safely placed in ROM.

literals Section (Type CODE)

The **literals** section contains all compiler-generated literal data. You do not have direct access to this data at the C language level. Compiler-generated literal data can always be safely placed in ROM.

const Section (Type CODE)

The **const** section is used for external or static variables that are explicitly initialized. These variables are declared using the keyword **const**.

In some cases, scalar static **const** variables might not be placed in this section. Instead, the compiler may decide not to allocate any storage for them and to substitute a constant value each time they are used.

If your application does not attempt to modify const variables, then you can safely place the const section in ROM. Direct assignment to const variables is prohibited by the compiler but can be achieved by using type casting or by using const inconsistently across different modules.

ioports Section (Type DATA)

The **ioports** section is generated whenever a program uses the **read**() or **write**() functions. The **ioports** section is used to hold the **_simulated_input** and **_simulated_output** variables that are used to facilitate I/O simulation with the XRAY Debugger. The **ioports** section should be placed in RAM.

vars Section (Type DATA)

The **vars** section is used for all static and external variables that are explicitly initialized. These variables are declared without the **const** keyword. Typically, this data is placed in RAM.

zerovars Section (Type DATA)

The **zerovars** section is used for all uninitialized external or static variables. These variables will be initialized to zero when the program is loaded into memory or when execution starts.

If a portion of a variable is explicitly initialized, then the entire variable is considered to be initialized and will be placed in the **const** or **vars** section. For example, in the declaration below:

```
char buffer [1024] = {0};
```

the first element of `buffer` is explicitly initialized so this variable cannot be placed in the **zerovars** section.

Variables in the **zerovars** section will be initialized to zero when the program is loaded into memory or when execution starts. Since the **zerovars** section contains only uninitialized variables, it does not take up any physical space in the executable file. The allocation and initialization of **zerovars** variables depend on the presence of a program loader:

- If the operating system has a program loader, then the loader must allocate memory space for the **zerovars** data. Some program loaders will initialize this memory to 0. If your loader does not provide this service, then your start-up routine should initialize this memory to 0.

- If the operating system does not have a program loader, then your program code must be permanently resident in ROM, and the **zerovars** data must be allocated to an address space within RAM. Your start-up routine should initialize the **zerovars** memory to 0 each time the program is restarted.

tags Section (Type DATA)

The **tags** section is generated when the **-Kt** compiler option is specified. This section contains symbol definitions to tag each function's entry and exit points. These tags may be used by real-time analysis tools to monitor program execution. The **tags** section should be placed in RAM.

Embedded Environments 9

This chapter gives you tips on using the Microtec C compiler to write embedded applications. It includes the following information:

- How to include assembler source code in your C program
- How to generate reentrant code
- How to write programs that use position-independent code and position-independent data
- How to modify the system functions and initialization routines included in the distribution
- How to reserve a register for shared data, system data, reentrant programs, or shared programs
- Tips on data initialization
- Tips on using the linker to initialize program variables in RAM at start-up time
- How to create a linker command file

Embedded Applications

An embedded system consists of a microprocessor and its associated peripheral hardware and software. The software provides all operating system functions such as input/output, memory allocation, scheduling, and interrupt handling.

Typically, an embedded system does not have a standard operating system, run-time loader, or disk. Instead, its program code and constants usually reside permanently in Read-Only Memory (ROM). The program starts execution at a memory location determined by the hardware whenever a **RESET** occurs (at power-up).

Considerations for Embedded Systems

You should consider the following when working with an embedded system:

- The hardware starting address must coincide with the main entry point (or code that ultimately jumps to the main entry point). For this reason, the sample initialization routine in the file **entry.c** might need to be modified.

If any modifications are made, **entry.c** must be recompiled, and the result-

ing object module must be linked with your program. (The new version must be loaded ahead of the Microtec C library in the linker command file, or the old version should be replaced by the new one in the library.) See the section *User-Modified Routines* in this chapter for more information on the initialization routine and how to modify it.

- Since an embedded environment is a unique one that has its own memory organization and its own I/O system, you may have to customize several run-time library routines to accommodate the environment. Those routines that might need to be modified include the I/O routines (**read**, **write**), the memory allocation routines (**malloc**, **calloc**), and the system functions (**open**, **close**).
- Initialized variables can be placed either in ROM or RAM or initialized by the program at run time. If variables are placed in ROM, they cannot be altered by the program. The *Data Initialization* section in this chapter discusses this topic in detail.
- To use the C I/O and memory allocation library, you must modify several routines to adapt them to your embedded environment. See the section *User-Modified Routines* in this chapter for more information.
- Interrupt handlers written in assembly language must save and restore all registers they use. These routines may call C functions as described in Chapter 7, *C Calling Conventions*, in this manual.

In-Line Assembly

Embedded applications require close and efficient control of the target architecture. To accommodate this need, the Microtec C compiler supports assembler statement intermixing with C code.

You can include any number of assembler lines by using the pseudofunction **asm** (which can also be specified as **ASM**).

Syntax:

```
asm([type[,]] [string [,string]...]);
```

Description:

type Tells the compiler what is returned by the **asm** invocation and, implicitly, where it is returned. By default, the value returned by the **asm** pseudofunction is of type **int**.

string Represents assembler instructions. Generally, each *string* separated by a comma corresponds to a new assembler line. You can insert as many assembler instructions as you like using a single **asm** statement as long as you follow these rules:

- Since each assembler statement must be on its own line, either use a separate string for each assembler statement or use **\n** to generate a newline character.
- Include at least one white space character (tab or space) in front of any assembler mnemonic.

The **asm** pseudofunction behaves from the outside exactly like a function (that is, it takes parameters and returns a type). However, **asm** does not generate a procedure call. Instead, it inserts your assembler code "in-line."

Like any C function, **asm** can be invoked with or without taking into consideration its returned value. The global optimizer in the compiler disables certain optimizations based on the returned value.

The **asm** pseudofunction cannot return structures or unions. For other types, the compiler expects the return value to be in the same register(s) as a function of that type. For more information about return values, see the section *Function Return Values* in Chapter 7, *C Calling Conventions*, in this manual.

Follow these guidelines when you add **asm** pseudofunctions to your code:

1. Write your C program.
2. Compile the program.
3. Look at the places in the assembler output where you are considering adding **asm** pseudofunctions.
4. Notice that the compiler adds underscores (_) in front of all variables and accesses auto variables by using **SP**.
5. Add assembly code (also known as "inserts") to **asm()** pseudofunctions within procedures.
6. Use **#pragma asm** and **#pragma endasm** to add assembly code outside procedures.

Examples

The following are some simple examples of how to use **asm** to put assembler lines into a C program.

Example:

```
main()
{
     asm(" move.l $1000,SP ; initialize stack pointer");
     ...
```

In the example above, the compiler inserts the quoted string immediately after the prologue code that it generates for `main()`. A newline is added after the string to avoid concatenating it with the next line.

Since no type was specified in the above **asm** statement, the return type is **int**. The return type is ignored since the return value is not assigned to any variable.

Example:

```
asm(char*, " MOVE.L #0,D0 " ) [100]='*';
```

This example shows how to impose a `char` array view on physical memory.

Example:

```
asm( " NOP"," TRAP #0");
asm( " NOP\n TRAP #0" );
asm( " NOP "," T" "RAP #0 " );
/* no ','after T - same as " TRAP #0 " */
asm( " NOP "); asm( " TRAP #0 " );
asm(int," NOP"," TRAP #0");
```

These statements all generate the same code:

```
NOP
TRAP #0
```

The compiler supplies an **End-of-Line** character at the end of every string argument. Thus, in the statement:

```
asm( " NOP\n"," TRAP #0");
```

the `\n` is unnecessary.

Examples:

```
asm("label:"); /*label is in col 1 */
asm("MOVE.L #1,4(A6,D2.W); /* ERROR - MOVE in col 1 */
asm(" ORG.S $100");
asm("thous DC.L $1000,$2000,$3000 " );
asm(" garbage in ...");
```

These five examples show how powerful (and potentially dangerous) the **asm** feature can be. Because the compiler passes the string arguments of **asm** directly to the assembler, it does no syntax checking of your assembler statements. Take great care when you use the **asm** pseudofunction.

Features of Assembler In-Lining

This section describes features of **asm** and provides examples.

Assigning asm to a Variable

You can assign the return value of **asm** to a variable, as you can with any other C function. The following example determines the value of the stack pointer.

Example:

```
sp_reg=asm(" move.l sp,d0  ; get value of sp register");
```

The above **asm** statement returns an integer (the default type). According to the calling conventions, integers (like other scalars) are returned in register `d0`. Since **asm** behaves strictly like a function, the compiler generates code to move the returned value to the target.

The compiler produces this sequence in the assembly file:

```
move.l sp,d0      ; get value of sp register
move.l d0,_sp_reg
```

Returning a Typed Value

As stated previously, you can use the type argument to **asm** to tell the compiler what is returned by the **asm** invocation and, implicitly, where it is returned.

Example:

```
int *sp=asm(int *, " move.l SP,D0");
```

In this example, since the compiler knows a pointer is returned in `D0`, the compiler will, after the pseudo-invocation of **asm**, generate code to move `D0` to the variable

SP.

Example:

```
double d=asm(double, " fmove FP1,FP0");
```

In this example, since floating-point values are returned in `FP0` (assuming a coprocessor), the value of `FP0` will be moved into `d`.

No real call to **asm** takes place. The passing of the parameters and the call itself are replaced by the string(s) that you supply. You should store in `D0` (or whatever the calling conventions dictate) the value that interests you.

Using #define for Readability

You can make **asm** statements more readable with the **#define** directive.

Example:

```
#define D4 asm(int, " move.l D4,D0")
#define D5 asm(int, " move.l D5,D0")
if (D4>D5)
...
```

This example shows that it is possible to access every single machine device at the C level. The overhead is confined to a move. For `DO (and` **FP0**`)`, not even this move is required.

Variable Names Inside asm

You can insert global and local variables by name inside an **asm** string. At the assembly level, a global variable is represented by prepending an underscore (_) to its name. For example, global variable **x** is represented by **_x**. Thus, when accessing a global variable inside an **asm** string, you can simply add the underscore.

Representation of local variables at the assembler level is more complex. They are represented by frame offsets or by registers that can change from one release of the compiler to the next. Their addresses can even change between two compiles if new code or variables have been added to a procedure.

The Microtec C compiler lets you insert variable names in any **asm** string by simply quoting them with a back quote (`). Since it is unlikely that this character will appear in any assembler statement, Microtec has chosen it as the default.

Example:

```
foo()
{
int automobile, garage;
     asm(" move.l `automobile`,`garage`");
     . . .
}
```

The actual addresses of the inserted variables will be supplied while conforming to the usual scope rules. According to the memory/register binding in force for the compilation, the above example will generate:

```
move.l 12(a6),d4
```

If you want to use a character other than the back quote, use the **-ui***char* option to change the insert character to *char*. Use the **-ui***char* option inside a **#pragma option** rather than from the invocation line because you do not want to change the insert character between compilations.

Inserts are recommended for global variables as well as local variables. Variable names inside a string are not affected by compiler options that modify naming conventions, like **-upd**, unless they are enclosed in back quotes. However, inserts are option-sensitive.

Example:

```
#pragma option -upd
/* prepend a dot to all global names */
. . .
asm(" move.l _global,_global2");
```

In the example above, the **-upd** option cannot affect the global names inside the string.

Example:

```
#pragma option -upd
/* prepend a dot to all global names */
. . .
asm(" move.l `global`,`global2`");
```

In this example, the inserts will reflect the `-upd` option.

#pragma asm or asm

Since **asm** is a pseudofunction, it can only be used where a normal function can be invoked, namely inside a procedure. Outside a procedure, any number of unquoted

assembler instructions can be inserted between the pair **#pragma asm** and

#pragma endasm. You can use these directives to create your own assembly procedure.

Example:

```
#pragma asm
#if _FPU
     fmove.l     ROUNDING,FPCR ; set rounding mode
#endif
#if _CHAR_SIGNED
 #if (_68020 || _68030 || _68040 || _CPU32)
   extb.l   D1
   move.l   D1,D0
 #else
   ext.w    D1          ; sign extend
   ext.l    D1
   move.l   D1,D0
   #define XXX signed.s
 #endif
#else
   moveq    #0,D0       ; zero extend
   move.b   D1,D0
   #define XXX unsigned.s
#endif
#include XXX
#pragma endasm
```

#pragma asm is more readable for long sequences of assembly language. You can use user-defined macros as well as predefined macros inside **#pragma asm/ #pragma endasm** pairs, since full preprocessor functionality is available. Assembler inserts (**asm** function) cannot be used.

You can specify the size of a **#pragma asm** and **#pragma endasm** procedure block using the **#pragma option** preprocessor directive and the **-Zi***number* command line option.

Example:

```
#pragma option -Zi10
#pragma asm
      . . .
#pragma endasm
#pragma option -Zi
```

In this example, `#pragma option -Zi10` sets the size of the `#pragma asm` and `#pragma endasm` procedure block to 10 bytes. The line `#pragma option -Zi` resets the size for subsequent **#pragma asm** and **#pragma endasm** blocks and **asm()** pseudofunction invocations back to the default (infinite size). For more infor-

mation on the **-Zi***number* compiler option, refer to Chapter 2, *UNIX/DOS User's*

Guide. For more information on the **#pragma option** preprocessor directive, refer to Chapter 3, *The Preprocessor*, in this manual.

Considerations for Assembler In-Lining

The content of any string is passed to the assembler as is. No control is placed on the string. Direct consequences of this are:

- You must be ready to read errors issued by the assembler.
- The compiler makes pessimistic assumptions on the size of the inserted instructions. Every jump crossing the instructions might be long. Consider the following example:

```
asm(" DS $1000")
```

- **asm** is not portable. It is likely, however, that a program which uses direct machine instructions was never meant to be portable.

  Example:

```
printf( "%x",asm() );
```

  This statement prints the contents of register **D0**. The data printed could be completely different if the program is compiled on another compiler.

- By default, the compiler uses the assembler command line flag **noabspcadd**. However, when PC-relative addressing is specified (**-Mdp**, **-Mcp**), the compiler uses **abspcadd**. This means that an absolute expression in conjunction with the mnemonic **PC** (e.g., **5(PC)**) refers to an absolute address accessed through PC-relative mode rather than to a relative displacement to the current program counter. Hence, **asm** strings that use this addressing mode could be assembled differently depending on the compiler options specified.

  For more information about the **abspcadd** assembler command line flag, refer to the Microtec *Assembler/Linker/Librarian User's Guide and Reference for the 68000 Family*.

When you use **asm**, check the effects of optimizations. Even though some optimizations are disabled, the Microtec C compiler takes quite a bit of freedom in rearranging, changing, and deleting code. This behavior could affect your program.

Addressing

A variety of methods exist for accessing addresses.

Direct Memory Addressing

You can examine and modify absolute memory locations by defining a macro that functions exactly like a normal C variable, except that it is located at a particular memory address given by a number. For example, a macro called **memloc** is defined in the following example. This macro will represent the contents of the byte at location 1A (hexadecimal). The macro definition is:

```
#define memloc(*(char far *)0x1A)
```

Whenever `memloc` appears in the program, it is replaced with the macro definition text `(*(char *)0x1A)`, which means "1A hexadecimal, treated as a pointer to a character and dereferenced." If `memloc` appears in an expression or on the right-hand side of an assignment, it represents the contents of byte location `0x1A`. If `memloc` appears on the left-hand side of an assignment, it represents the byte address `0x1A`.

The macro **memloc** can be used like an ordinary byte variable:

```
chrl = memloc + 1;
memloc = 'a';
```

The macro **memloc** can also correspond to a memory-mapped I/O location, in which case **memloc** must be **volatile** since each use would have the side effect of reading or writing:

```
#define memloc (*(volatile char *)0x1A)
```

Calling a Function at an Absolute Address

Use the **asm** pseudofunction or assembly language to call a function at an absolute address (such as a system routine). However, if you are concerned about porting your program to a different processor, use the technique described here.

This section describes how a C program can call an independently compiled and linked function that is placed at an absolute address without having to use assembly language. You can use this technique when an application program is required to call a system routine at a fixed ROM address.

To call a function at an absolute location, you can define a macro that declares a function.

Example:

```
#define ABS_FUNC (*(int (*)())0x124)
```

The macro defines `ABS_FUNC` to be an integer-valued function at location `0x124`. You can then call `ABS_FUNC` just like an ordinary function:

```
i = ABS_FUNC(); /* ABS_FUNC can be in any expression */
```

If parameters are being passed, the routine you are calling must have a compatible calling sequence. The return value for integer functions is placed in register **D0**. If a return value is expected, this is where the called function must place it. For more information on function parameter passing and return values, see Chapter 7, *C Calling Conventions*, in this manual.

Reentrant Code

A routine is reentrant if it can be interrupted during its execution and reinvoked by subsequent calls (for example, from an interrupt service routine) any number of times. When each subsequent call has completed its invocation, the prior invocation resumes processing where it left off without loss of data.

The code generated by the compiler will be reentrant provided you follow these rules:

- Do not write to global variables.
- Do not write to local static variables.
- Do not perform any noninterruptible tasks such as certain I/O operations.
- Use the non-reentrant library routines only in the manner described below.

For the purposes of this discussion, a thread of computation, or simply “thread,” is a given sequence of instructions. A thread can be interrupted by an external interrupt, and control can be given to another thread at almost any time. Multiple threads may access the same executable code at any given time. An interrupt service routine, while it is active, is also a thread of computation.

Interrupt service routines, when they are interrupted and reentered by another interrupt, constitute multiple threads of computation. In a multiprocessing or multitasking environment, each process or task is a thread of computation. A reentrant routine is one that will function correctly even though multiple threads may be executing it simultaneously.

In general, the I/O routines that are declared in **stdio.h** are non-reentrant because they modify elements of the array **_iob**, which is declared in **stdio.h**. However, the three string I/O functions **sprintf**, **vsprintf**, and **sscanf** are reentrant. To make file I/O reentrant, each task should allocate its own file structure buffer on its stack

instead of calling **fopen**. Therefore, the following code can be used for writing formatted output:

```
i = sprintf(buff, format_string, arguments);
write(1, buff, i);
```

Description:

`sprintf`	Returns the number of characters written.
buff	Is an area of memory where the formatted output will temporarily be written.
format_string	Is a **printf** format string.
arguments	Is a list of arguments used by *format_string*.
`write`	Outputs *i* bytes from *buff* to the output port.
`1`	Causes `write` to write to **stdout**.

There is another approach to using I/O routines when reentrancy is required. If each thread performs I/O through a unique **FILE *** or "stream," the I/O routines can be considered reentrant. This is due to the fact that each thread is modifying its own unique element of the **_iob** array. However, because **fopen** must scan the **_iob** array to find an unopened element, each stream must be opened in the application's start-up code before individual threads are activated.

Functions that modify **errno** are also non-reentrant. Many of the functions declared in **math.h** modify this variable, along with the functions **strtod**, **strtol**, **strtoul**, and others. If a multi-threaded environment is required, the easiest way of using these routines is simply not to use or modify **errno** in any user-written code.

The other functions listed as non-reentrant in Chapter 5, *Library Extensions*, in this manual may be used in a multi-threaded environment; however, only one thread may call any group of functions accessing the same static variable. For example, only one thread may call the functions **malloc**, **calloc**, **zalloc**, **realloc**, and **free**. These routines may be called by multiple threads only by using one of the techniques listed in the following sections.

A Simple Solution

In situations in which multiple threads must call non-reentrant library functions, there is a solution that allows this to be done. During a context switch, the variables in question are saved. When the thread is resumed, the saved data is restored. This

approach requires a memory storage area for each thread although an interrupt service routine can save and restore the data from the stack. Also, if a large amount of

data must be stored, the save and/or restore operation can take a prohibitively large amount of time.

Multitasking/Multi-Threaded Environments

There are two common types of applications that use multitasking or multi-threaded execution. In the first type, there is a single set of code that serves a number of different tasks or execution threads. An example of this would be a communications controller that uses the same code to service a number of communication ports. The second type has different code for each of the tasks. An example of this second type would be a laser printer, where one task manages communication and another task manages the print engine. Naturally, there are also applications that have aspects of both styles.

A multi-threaded application frequently needs to access memory that is shared by all of the threads (i.e., global memory) as well as memory that is used only by one specific thread, often called thread-local storage. Additionally, each thread needs its own stack space. In many cases, allocating the thread-local and stack memory is handled by an operating system. Access to shared variables needs to be done with care. If the variable is only read by the thread, there is no problem, but if one thread may update the shared variable while another is reading it, there are opportunities for race conditions and bugs that are very difficult to locate. These accesses should be controlled by semaphores or monitors.

MCC68K provides the **-Md***n* option, which forces all memory access to be relative to **A***n*. This is used for all thread-local storage access. Most often, **A5** is used for this purpose. When the thread is created, **A5** is loaded with the address of a block of memory that contains the thread-local memory. The functions that make up the thread are compiled with this option.

The **-Kh***reg* option is used to prevent the compiler from generating code that uses the specified register. Files that contain functions shared by all threads should be compiled with the **-KhA5** option.

Thread-local data must be kept separate from shared data. This means that the data sections for the thread-local and shared data must not have the same names. This is done by specifying unique names for the **vars** and **zerovars** sections using the **-NZ** and **-NI** options, either for the compilations for the thread functions or for the shared functions.

Since the data in the thread-local sections is addressed as an offset from **A5**, the linker is instructed to allocate these sections at location zero in memory, although the actual location will be somewhere else. When the thread is created, a block of memory is allocated that is large enough to hold the **vars** and **zerovars** sections of

memory is allocated that is large enough to hold the **vars** and **zerovars** sections of thread-local memory.

Often it is easiest to avoid creating an **vars** section by not using any static data. Otherwise, the **tvars** section will need to be initialized when the thread is created. The **INITDATA** section will contain a copy of the initialized values for the **initivars** sections that are specified in the linker command **initdata**. Although it is possible to have the linker create initialization data for both global data and thread-local data, these are not identified in the **INITDATA** section with the target section names. As a result, you will not be able to use the *initcopy* routine, since this would copy values over the "virtual" location of the **vars** section, not the actual location. You can use the *initcopy* routine if you have only thread-local data and do not want to initialize any global data. Since this is usually inconvenient, most applications will initialize variables in executable code when the thread starts.

Access to global variables can be done in two different ways. The preferred way is to create shared functions that are used to obtain or modify the values of global variables. These routines can contain the semaphore code necessary to insure that the value is always consistent. The less preferred method is to have a shared function that returns the address of the global variable to the thread. This should only be used if the variable is only read by the thread.

Here are two examples of how to build a multitasking/multi-threaded application. The first example shows how to build an application that has one set of functions that are shared by several threads; the second shows an application with different code for each thread.

First Approach

In the first solution, thread functions are compiled using **A5**-relative addressing. The **vars** and **zerovars** sections are renamed and the **A5** register is reserved for thread-local access. The linker command files force all user data to be placed into absolute locations in memory starting at location zero. These modules are then linked to the appropriate **A5**-relative addressing library. Memory for library functions is allocated dynamically at the beginning of each thread, and the **A5** register is set to address this space. Follow these steps:

1. Compile all user-written routines that are to be shared between threads using **A5**-relative addressing. Use of static data should be avoided in these routines. If static data is used, it must be uninitialized. The **vars** section must be empty; the **INITDATA** command of the linker will not initialize this data correctly. Use these options:

   ```
   -Md5 -Xp
   ```

2. User modules that are not shared between threads must be compiled for absolute addressing; the **A5** register is reserved and the **vars** and **zerovars**

sections are renamed. Weak externals must not be used in these modules. This may be done by using the options:

```
-Kha5 -Xp -NZuserzero -NIuservars
```

Unless that variable is intended to be used in inter-thread communication, no task should use a static variable name that is the same as a static variable name used by another thread.

3. Modify the code in **entry.c** to dynamically allocate memory for each thread for the **zerovars** section created by the libraries and any user data created in Step 1. If a thread requires stack space, it must be allocated at this time also. Set nonstack memory to zero at start-up time.

   The sample code in **entry.c** for reentrant library support shows how memory might be allocated for one thread.

4. If static data from a library function or a user-written shared function must be accessed by a function compiled with absolute addressing, one of the following techniques may be used:

   - Compile a function that returns the address of the data in question. Compile the function as in Step 1 using A5-relative addressing. Call this function from the function compiled with absolute addressing to determine the address of the data.
   - Use a macro that inserts assembly code using A5-relative addressing. A macro to access **errno** might be written as follows:

     ```
     #define errno (*(ASM (" xref `errno`"),\
     ASM (" lea (`errno`).w(a5),a0"),\
     ASM (int *," move.l a0,d0")))
     ```

5. Do not use the macro version of any I/O routine. Use the function version of an I/O routine by placing the function name in parentheses.

6. If the **INITDATA** feature of the linker is to be disabled, set the command line option **-D_EXCLUDE_INITDATA** to prevent **csys.c** from initializing user static data.

7. When linking, do not use the default linker command file. The wrong library will be linked if you do. Use the **LOAD** command to load the correct **A5**-relative addressing library. Use the **-e** option to specify the

linker command file. The following is a good starting point for writing a linker command file:

```
listabs       publics,internals
listmap       publics
format        ieee
extern        ENTRY
index         ?A5,zerovars,$8000
                    ; entry.c will point A5
                    ; to section zerovars
initdata      uservars
sectsize      heap=$8000
sectsize      stack=$1000
sect          zerovars=0
sect          literals=$10000

order   zerovars
  order literals,strings,const,initfini,code    ; ROM
  order ??INITDATA                              ; ROM
  order userzero,uservars,ioports,heap,stack    ; RAM
  load  /usr/mri/lib/mcc68ka5XXX.lib            ; library
```

In this example, **zerovars**, used by the thread, is allocated at location zero. The actual location will be wherever memory is allocated when the thread is initialized. Since there may be several sections allocated at location zero, the linker may issue warning messages. Carefully verify that these messages are the result of overlapped thread-local data sections.

8. At run time, when a context switch occurs, the **A5** register is set to address the data space of the thread that is about to run.

Figure 9-1 shows the memory configuration used by this technique.

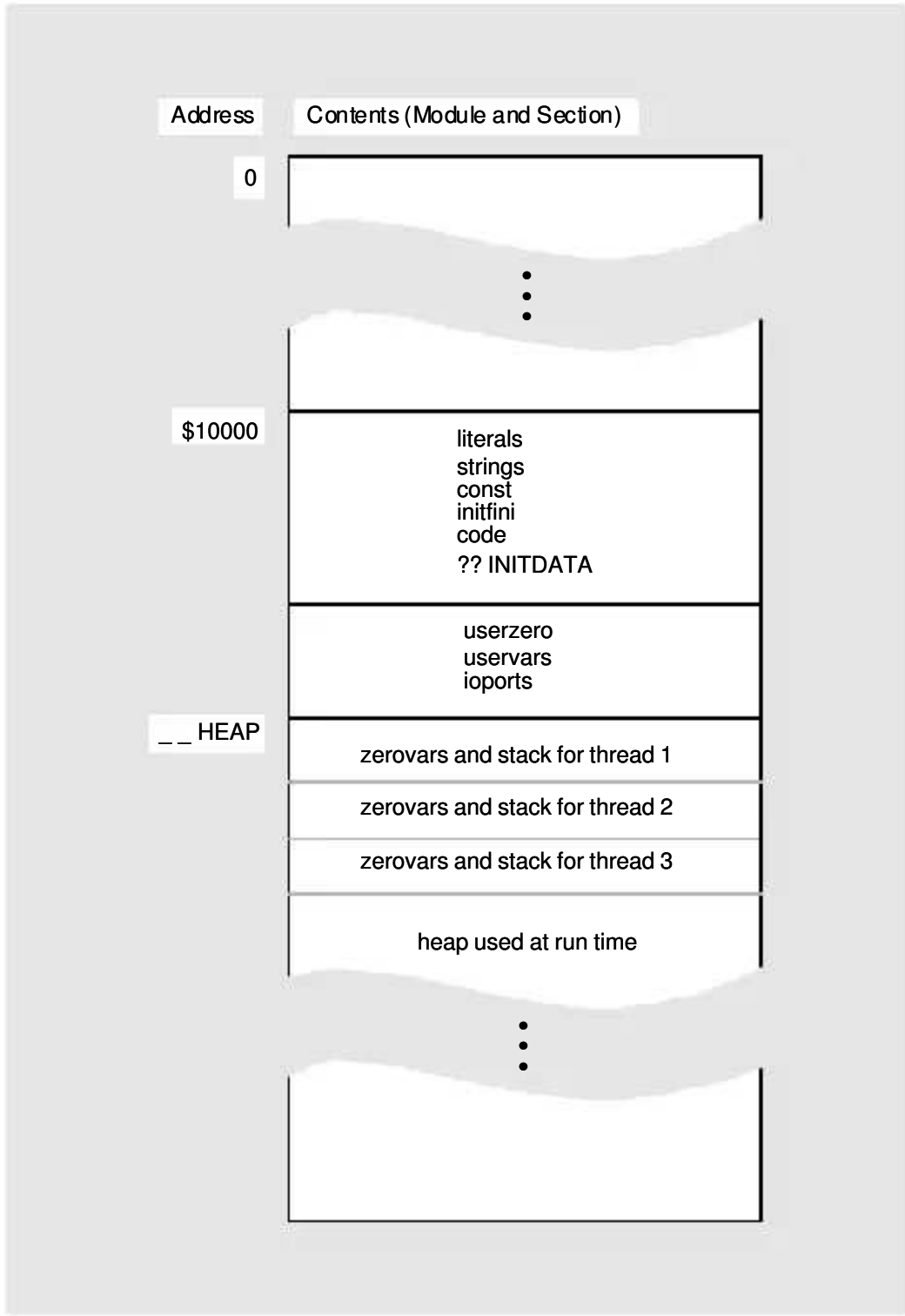
Address
Contents (Module and Section)
0
$10000
literals
strings
const
initfini
code
?? INITDATA
userzero
uservars
ioports
_ _ HEAP
zerovars and stack for thread 1
zerovars and stack for thread 2
zerovars and stack for thread 3
heap used at run time

Figure 9-1. Memory Configuration for Multi-Threaded Environments (Approach 1)

Second Approach

In the second solution, the approach is to compile all thread functions using **A5**-relative addressing. Initialized static data is not used, so the **vars** section is empty. The **zerovars** section is renamed to a name that is unique to each thread. At link time, these sections are then positioned to begin at the same memory location. At run time, memory is allocated for the data in each thread in your start-up routines and the data is referenced by **A5** addressing. Follow these steps:

1. Compile all your modules using **A5**-relative addressing and rename the **zerovars** section to a name unique to each thread. Do not define any initialized data in these modules as the **INITDATA** feature of the linker does not work well with this technique. In other words, the **initvars** section must be empty for all these modules. There should be no common static variable names between threads. Weak externals must not be used. Compiler options that might be used to do this are:

   ```
   -Md5 -Mcp -NZname -Xp
   ```

2. Modify the code in **entry.c** to dynamically allocate memory for each thread for both a **zerovars** section created by the libraries and the user data sections created in Step 1. If a thread requires stack space, it must be allocated at this time also. Set nonstack memory to zero at start-up time.

3. Link to the appropriate **A5**-relative addressing library. Set the starting point address of each data section created in Step 1 to the same value. The linker will generate **Section Overlap** error messages caused by the placing of multiple sections in the same memory space, but these may be disregarded. The following can be used as a good starting point for writing a linker command file:

   ```
   listabs       publics,internals
   listmap       publics
   format        ieee
   extern        ENTRY
   index         ?A5,zerovars,$8000
                              ; entry.c must set A5
                              ; to section zerovars.
   sectsize      heap=$8000
   sectsize      stack=$1000
   sect    zerovars=0         ; position zerovars sect.
   sect    task1zero=554      ; position data for each
   sect    task2zero=554      ; thread at the same memory
   sect    task3zero=554      ; location.
   sect    literals=$10000
     order literals,strings,const,initfini,code    ; ROM
   ```

```
order literals,strings,const,initram,code    ; ROM.
order ioports,heap,stack                     ; RAM.
load  /usr/mri/lib/mcc68ka5XXX.lib        ; library.
```


4. At run time, when a context-switch occurs, the **A5** register is set to address the data space of the thread which is about to run.

5. The routines **_sbrk** (**s_sbrk.c**) and **_chkstk** (**s_chkstk.c**) perform run-time checks for heap and stack overflows. These routines will work only when the stack is located adjacent to the heap and at a higher address. When this arrangement does not occur, you must not use the **-Ks** command line option, and you must modify **_sbrk** to change the overflow checking algorithm.

Figure 9-2 shows the memory configuration used for this technique.

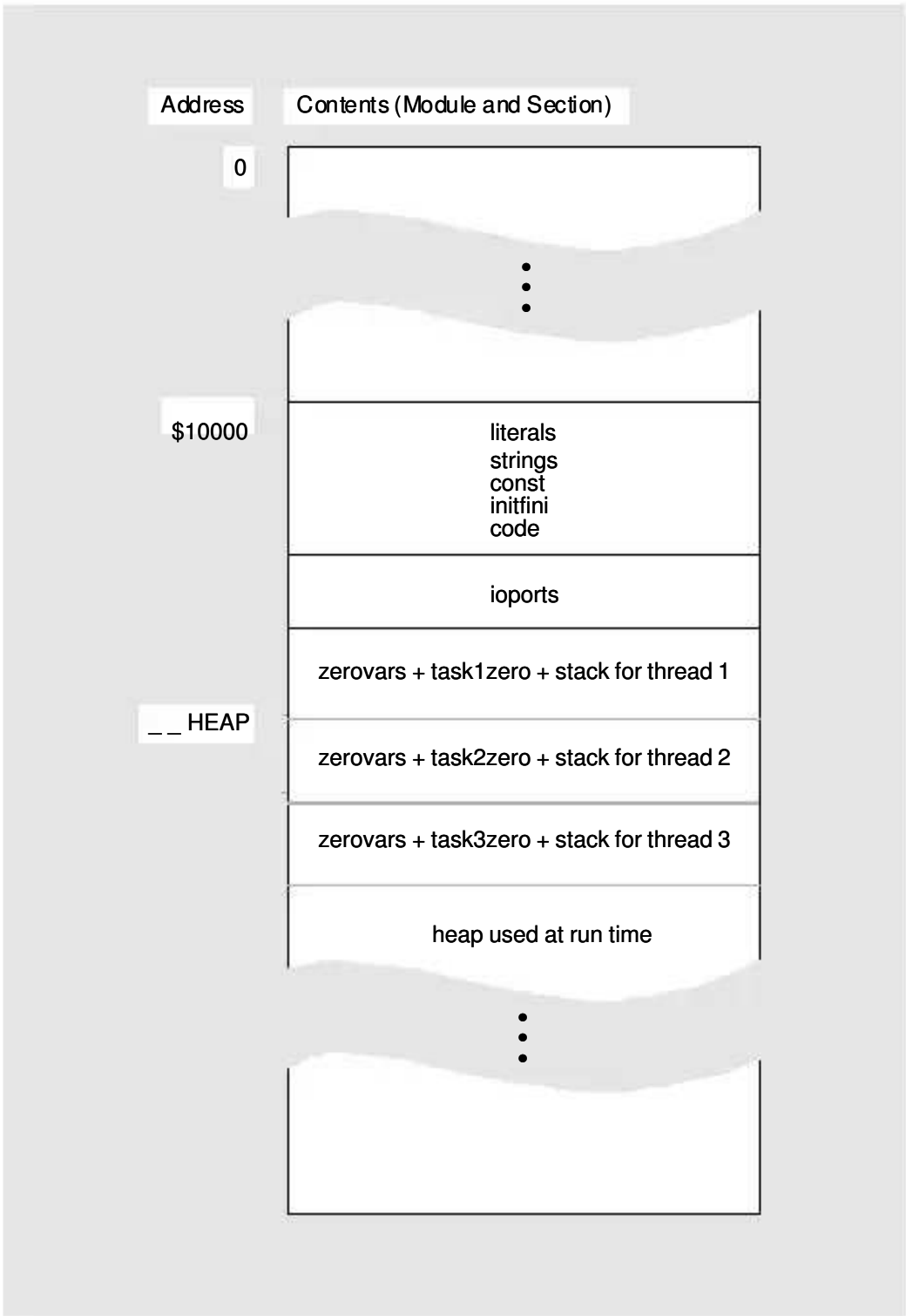
Address
Contents (Module and Section)
0
$10000
literals
strings
const
initfini
code
ioports
zerovars + task1zero + stack for thread 1
_ _ HEAP
zerovars + task2zero + stack for thread 2
zerovars + task3zero + stack for thread 3
heap used at run time

Figure 9-2. Memory Configuration for Multi-Threaded Environments (Approach 2)

Position-Independent Code and Data

This section describes position-independent code, position-independent data, and how they relate to the Microtec C compiler, the C programming language, and the 68000 family microprocessor.

By default, the compiler generates references to absolute code and absolute data. However, the command line options shown in Table 9-1 let the compiler generate references to position-independent code and position-independent data.

Table 9-1. Position-Independent Code Options

UNIX/DOS Option	Meaning
-Mcp	Generates position-independent code by using program counter-(PC) relative addressing for all references to code
-Mdp	Generates position-independent data by using program counter-(PC) relative addressing for all references to global data
-Md*n*	Generates position-independent data by using **A***n*-relative addressing for all references to global data (**A***n* is **A2**, **A3**, **A4**, **A5**, or **A6**)

For more information about command line options supported by the compiler, refer to section *Generate Position-Independent Code and Data* in Chapter 2, *UNIX/DOS User's Guide*.

Position-Independent Versus Position-Dependent

Position-independent code and data references are used for multitasking systems and for other programs that need to be loaded at different addresses for different executions. A program that is position independent can be loaded by the operating system at any available memory location and will work regardless of where it is placed in memory. However, a stand-alone program targeted for a specific CPU board and memory configuration does not need to be position independent because it is always loaded at the same memory location.

A position-independent program specifies its code and data addresses relative to a register by using register-relative addressing. The **-Mdp** option uses the program counter as the base register. The **-Md***n* option uses **A2**, **A3**, **A4**, **A5**, or **A6** as the base register.

The effective address is generated by adding a displacement (or offset) to the base. The base displacement is a two- or four-byte value that immediately follows the operation code in the instruction. This procedure for generating the effective address works because the relative positions of instructions and data in a program remain the same even though their absolute addresses can change each time the program is loaded.

Compiler Considerations

The compiler adheres to the following rules when generating position-independent code references **-Mcp**:

- The compiler does not use immediate addressing to load address constants.
- To load the address of a function, the compiler uses the PC-relative form of the Load Effective Address (**LEA**) instruction.
- To push the address of a function, the compiler uses the PC-relative form of the Push Effective Address (**PEA**) instruction.
- The compiler uses PC-relative addressing for all jumps and internal subroutine or function calls.

The compiler adheres to the following rules when generating position-independent data references **-Mdp** and **-Md***n*:

- The compiler does not use immediate addressing to load address constants.
- To load the address of a global data item, the compiler uses the register-relative form of the Load Effective Address (**LEA**) instruction.
- To push the address of a global data item, the compiler uses the register-relative form of the Push Effective Address (**PEA**) instruction.
- The compiler uses register-relative addressing for all references to static or global variables.

> **Note**
>
> Local automatic variables are already position independent because they are located on the stack and are addressed relative to the stack pointer.

Programmer Considerations

To generate-position independent code, you must adhere to the following rules:

- Do not call or jump to absolute code addresses within the program.
- Do not use pointers initialized to absolute code addresses within the program.
- Do not position a data reference and its target too far away from each other. The target of a data reference must be in the range of -32768 to 32767 bytes or -2147483648 to 2147483647 bytes if you use the **-MI** option with 68000 family processors having a default alignment of 4. For more information about these options, refer to section *Generate Position-Independent Code and Data* in Chapter 2, *UNIX/DOS User's Guide*.
  For example, a call at address 0 (whose displacement is at address 2) can only reference an address up to 32769, but a call at address 5000 (whose displacement is at address 5002) can reference an address up to 37769.

  The subroutine call at location 40000 (see Figure 9-3) cannot be made position independent because it attempts to reference an address that is not in the range of 7234 to 72769. The range is calculated using the following equations:

  40002 - 32768 = 7234
  40002 + 32767 = 72769

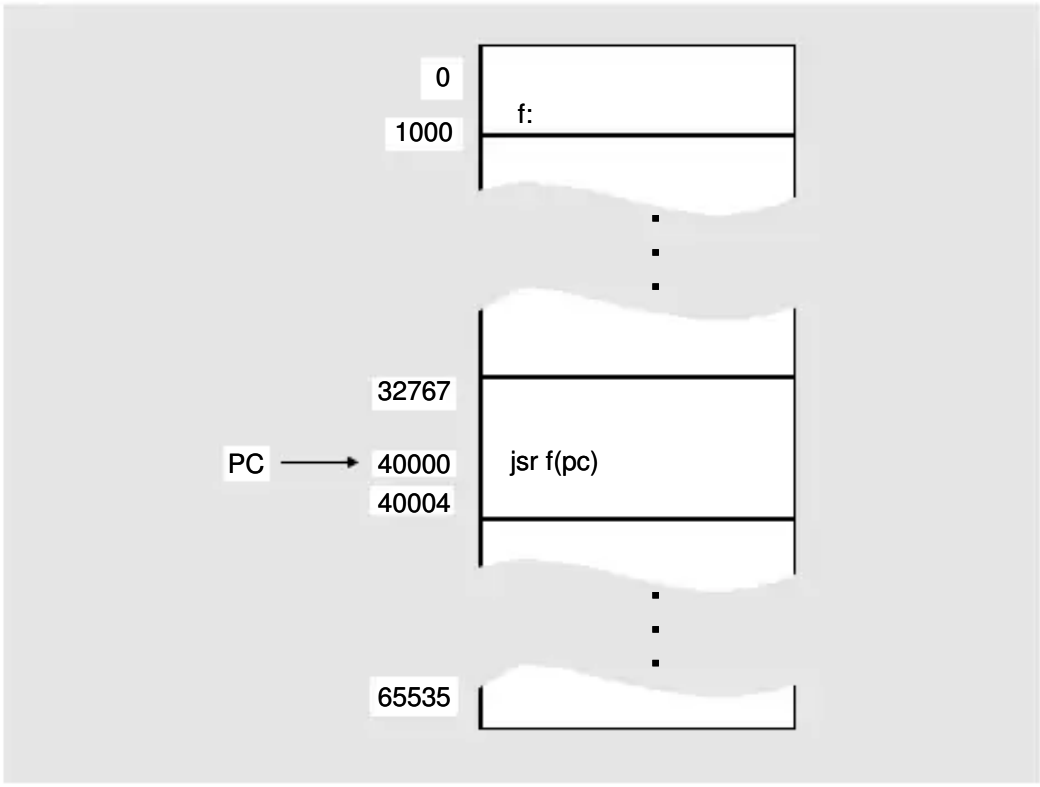

Figure 9-3. Position-Independent Code (Example 1)

To generate position-independent data, you must adhere to the following rules:

- Do not make data references to absolute data addresses within the program. For example, the following code cannot use register-relative data addressing because it uses an absolute data address:

```
/* reference to absolute data address */
read_from(0x02);/* read from data address 0x02 */
```

- Do not use pointers initialized to absolute data addresses within the program. For example, the following code fragment cannot use register-relative data addressing because it initializes pointers to absolute data addresses:

```
/* pointers initialized to absolute addresses of
strings */
char *table[3] = {"table", "of", "strings"};

/* pointer initialized to absolute address of data */
char c;
char *p_c = &c;
```

```
char *p_c = &c;
```


- Do not position a data reference and its target too far away from each other. The target of a data reference must be in the range of -32768 to 32767 bytes, or -2147483648 to 2147483647 bytes if you use the **-MI** option with 68000 family processors having a default alignment of 4. For more information about these options, refer to section *Generate Position-Independent Code and Data* in Chapter 2, *UNIX/DOS User's Guide*.

  For example, an instruction at address 0 (whose displacement is at address 2) can only reference an address up to 32769, but an instruction at address 5000 (whose displacement is at address 5002) can reference an address up to 37769.

  The instruction at location 40000 (see Figure 9-4) cannot be made position independent because it attempts to reference an address that is not in the range of 7234 to 72769. The range is calculated using the following equations:

  40002 - 32768 = 7234
  40002 + 32767 = 72769

  If you are using **A***n*-relative data, the default linker command file must be modified.

- The default linker command file contains the following:

```
; if using A5-relative data addressing,
; enable the next 2 lines:
;
; index ?a5, vars ; A5 will be pointed to sect vars
; load mcc68ka5XX.lib
```

Follow these steps to make your own command file:

1. Copy the default linker command file to your directory.
2. Uncomment the `index` and `load` commands.
3. Specify the correct library and path (see section *Libraries* in Chapter 2, *UNIX/DOS User's Guide*, for the location of the libraries on your host).
4. Invoke **mcc68k** with the **-e** option (see *Pass Command File to Linker (Driver Option)* in Chapter 2, *UNIX/DOS User's Guide*).

Note

Refer to the section titled *Code Organization* in Chapter 8, *Run-Time Organization*, in this manual for information on the location and addressing of the various code and data sections.

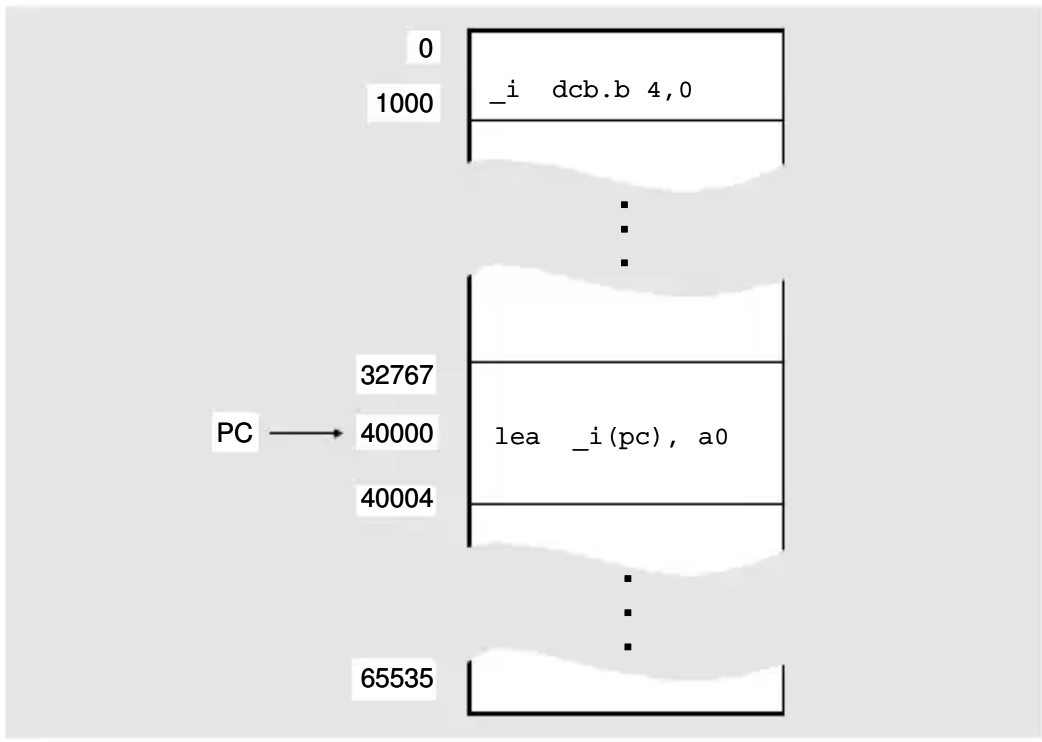

Figure 9-4. Position-Independent Code (Example 2)

Assembler and Linker Considerations

The assembler and linker automatically handle position-independent code; therefore, no special assembler or linker command line options need to be specified. When code is made position-independent, the resident run-time linker adds the address assigned to the first instruction to all the load addresses in the executable allowing the program to execute at these new locations.

Absolute Versus Register-Relative Addressing

The following program fragment example is used to show the different assembly language output generated by the compiler depending on the **-Mcp**, **-Mdp**, and **-Md***n* options.

```
int data_1; /* data_1 is a global variable */
main()
{
     function_1();
}
function_1()
{
     data_1 = 3;
}
```

The code generated as shown in Table 9-2 is position dependent when absolute addressing is used; it is position independent when register-relative addressing is used.

Table 9-2. Absolute Versus Register-Relative Addressing

Position-Dependent Code (CODE=ABS and DATA=ABS)	Position-Independent Code (CODE=PC and DATA=PC)

`...`
`L1: jsr _function_1`
`...`
`_function_1:`
`...`
`L2: moveq #3, d0`
`move.l d0, _data_1`
`...`
	`...`
	`L1: jsr _function_1(pc)`
	`...`
	`_function_1:`
	`...`
	`L2: lea _data_1(pc), a0`
	`move.l #3, (a0)`
	`...`

Absolute Addressing

Figure 9-5 shows absolute addressing for code and data. The **JSR** (jump to subroutine) instruction uses an absolute code address, and the **MOVE.L** (move long) instruction uses an absolute data address. The absolute addresses immediately follow the operation codes in memory. Absolute addresses are also used as effective addresses.

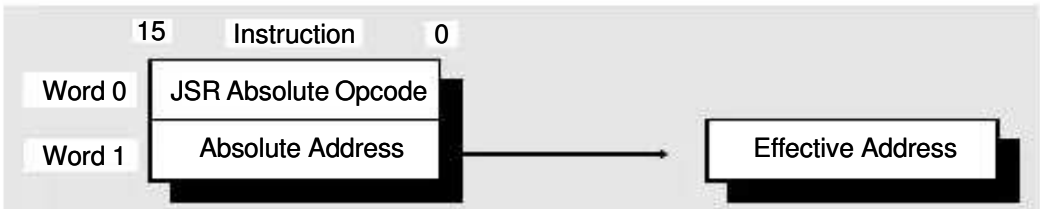

Figure 9-5. Absolute Addressing for Code and Data

Figure 9-6 shows how absolute addressing will position code in memory. Table 9-3 shows the commands for this example. The base address (**BA**) is the address where the program is loaded (i.e., fixed at link time). When the target of the **JSR** is computed, the Program Counter (**PC**) contains the address of the word containing the absolute address.

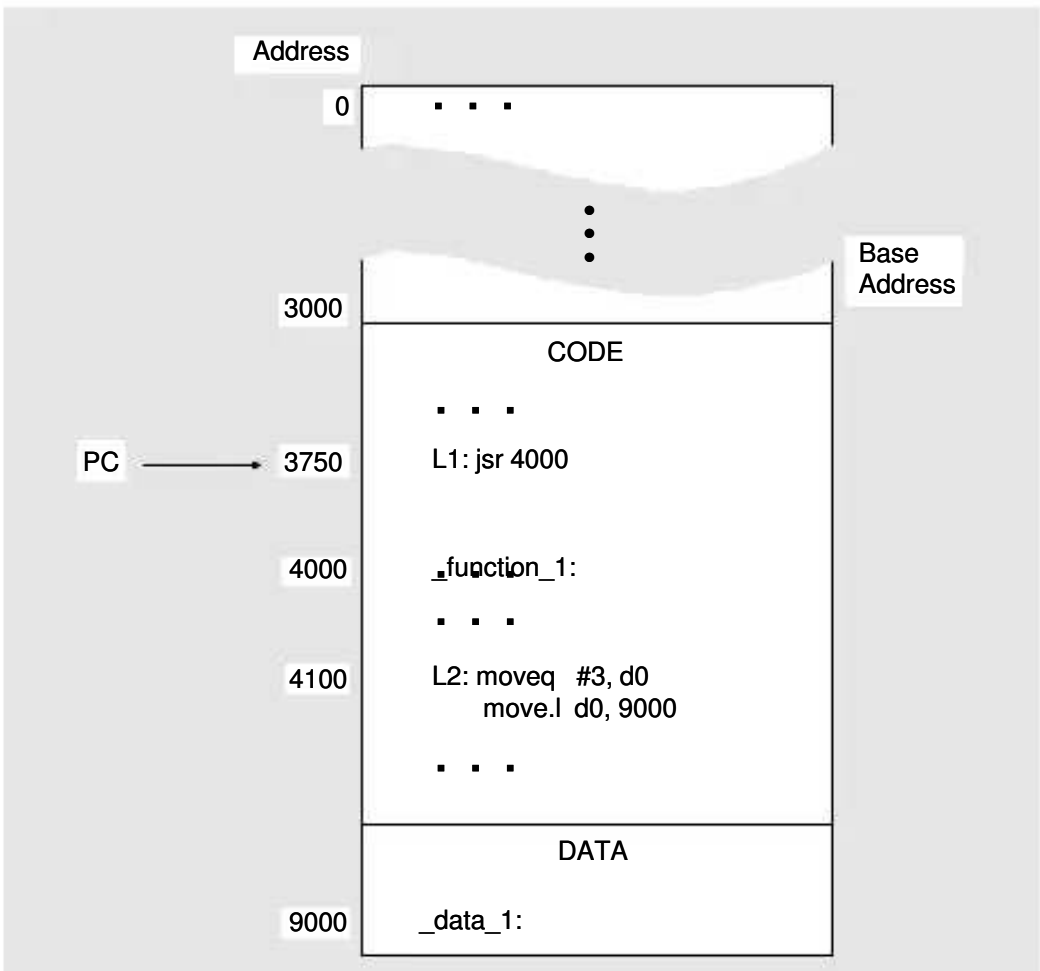

Figure 9-6. Absolute Addressing Example

Table 9-3. Commands for Absolute Addressing Example

PC	Description
3750 = `L1`	Calls `_function_1:` `jsr _function_1`
4000 = `_function_1`	The next section of code ". . ." is executed

4100 = `L2`	The value 3 is assigned to `_data_1:`
	`moveq #3,D0`
	`move.l d0,_data_1`

PC-Relative Addressing

Figure 9-7 shows PC-relative addressing for code and data. The **JSR** (jump to subroutine) instruction uses a PC-relative code address, and the **LEA** (load effective address) instruction uses a PC-relative data address. The effective address is determined by adding the contents of the program counter register to the displacement (or offset). When the target of the **JSR** is computed, the program counter contains the address of the word containing the offset.

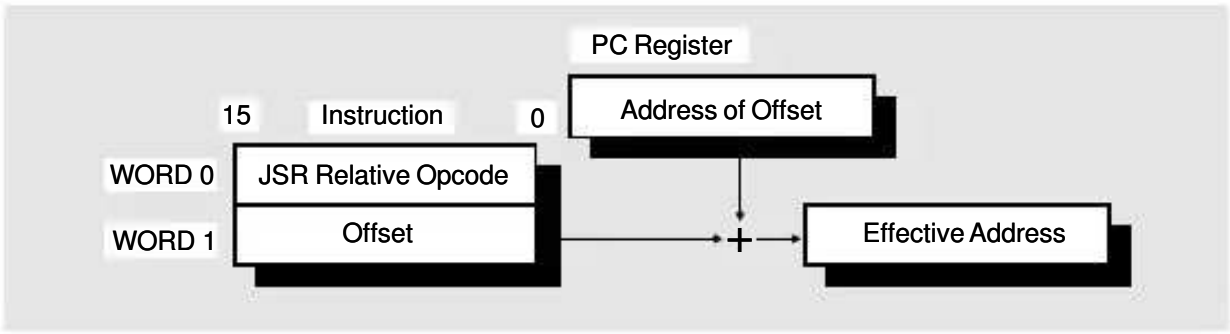

Figure 9-7. PC-Relative Addressing

Figure 9-8 illustrates how PC-relative addressing will position code in memory. Table 9-4 shows the commands for this example. The base address is the address where the program is loaded (i.e., determined at load time). Regardless of where the program is loaded, the relative positions of code and data remain the same.

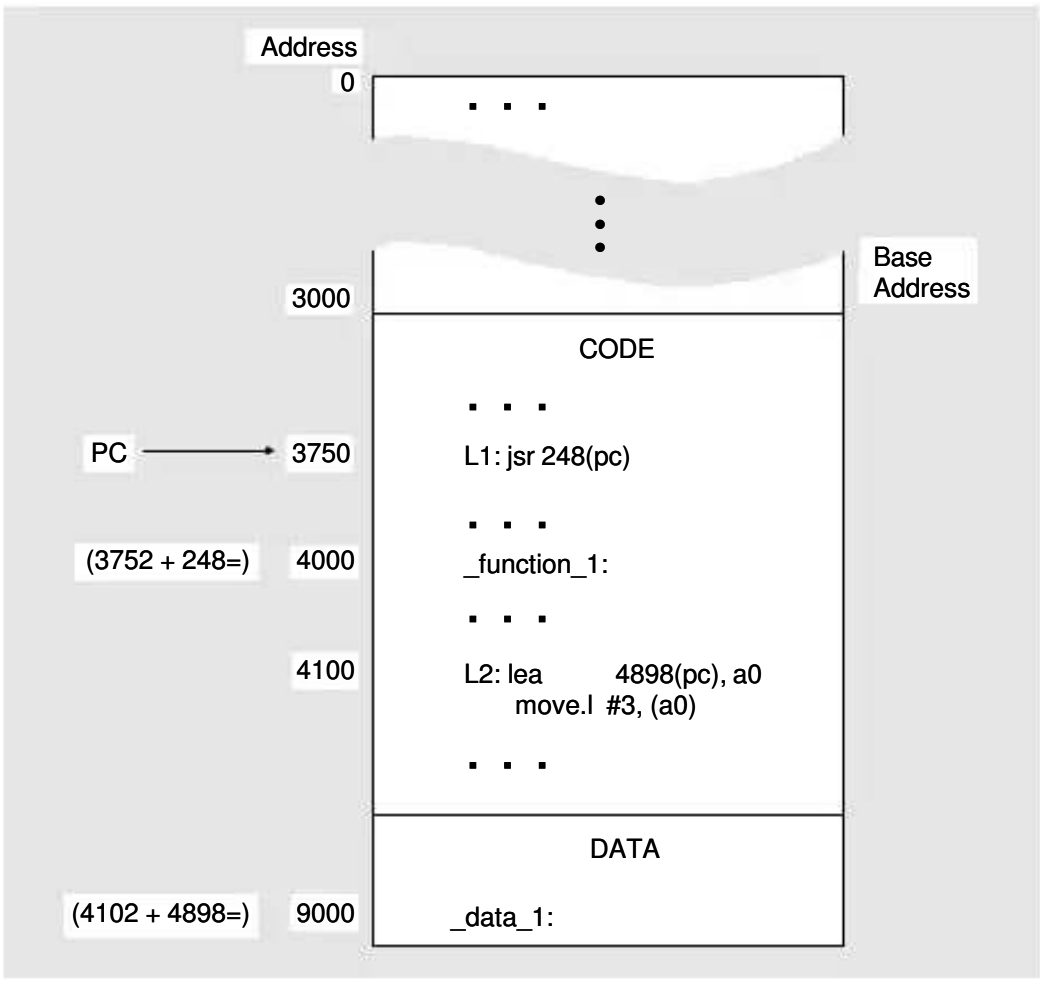

Figure 9-8. PC-Relative Addressing Example

Table 9-4. Commands for PC-Relative Addressing Example

PC	Description
3750 = `L1`	`Calls _function_1:` `jsr _function_1(pc)`
3752 + 248 = 4000 = `_function_1`	The next section of code "..." is executed

4100 = `L2`	The value 3 is assigned to `_data_1`:
	`lea _data_1(PC),a0`
	`move.l #3,(a0)`

User-Modified Routines

This section describes the routines that you must modify when using the Microtec C compiler in certain environments. Sample versions of these routines are provided on the distribution, but it is assumed that these sample routines will be modified extensively or used only as examples for writing your own routines.

The first part of this section covers the routines needed for users running on either an embedded system or a non-UNIX system.

User-Modified Routines for Embedded Systems

If you are running on a non-UNIX or embedded system, the routines **inchrw**, **outchr**, **entry**, **_START**, and the functions **sbrk** and **_exit** must be modified to work with your embedded system environment. These routines are described in the following sections. Simple versions of these routines and functions are provided in distribution files. These sample routines will work with the XRAY Debugger and are sufficient for initial debugging in that environment.

In Character Routine

The **inchrw()** routine reads characters from the **_simulated_input** variable. The XRAY Debugger **INPORT** command may be used to specify the source of input data for **_simulated_input:**

```
inport &_simulated_input [,input_source]
```

If no **INPORT** command is issued, the XRAY Debugger will read from the standard input device.

inchrw() returns an integer with a value between 0 and 255, or -1 if an error occurs.

Out Character Routine

The **outchr(ch)** routine writes characters to the **_simulated_output** variable. The XRAY Debugger **OUTPORT** command may be used to specify the destination of the output data for **_simulated_output**:

```
outport &_simulated_output [,output_destination]
```

If no **OUTPORT** command is issued, the XRAY Debugger will write to the standard output device. Its return value is ignored.

Start Routine

Start Routine

The start routine (**_START**) does the following:

- Initializes **_randx**.
- Completes initialization of the heap.
- Initializes the I/O system.
- Opens the standard files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**.
- Initializes global variables.
- **_simulated_input** and **_simulated_output** are placed in the **ioports** section. This section should be located in RAM.
- Can call the **initcopy** routine for copying initialized data in the **vars** section from ROM to RAM (see *Using the Linker* in a later section of this chapter for a description of the LNK68K **INITDATA** command).
- Calls your **main** function with the three arguments **argc**, **argv**, and **_environ**.
- Calls **exit** whenever your **main** function returns. The **exit** function flushes and closes all opened files and then calls **_exit** to terminate the program.

Exit Routine

A simple version of the **_exit** routine is included in the file **entry.c**. The **exit** routine is implicitly called at the end of each **main()** function to return control to the operating system. In this implementation, **exit** closes all opened files and then calls **_exit**.

Initialization Routine

The initialization routine initializes the embedded system environment. It should meet the following requirements:

- Be written in assembly language.
- Set up values for the stack pointer.
- Provide a static variable for the heap pointer.
- Call the **_START** routine.

The sample initialization routine (**entry**) is the main entry point. It does the following:

- Initializes **SP** as **.STARTOF.(STACK) + .SIZEOF.(STACK)**, which will

be defined at link time.

- Initializes __**HEAP** as the starting address of the **heap** section, which is located after all other sections.
- Initializes the frame pointer.
- Clears the **zerovars** section.
- Sets the **a5** register for **a5**-relative data addressing (optional, done in **mcc68ka5***xx***.lib**).
- Calls the C initialization routine **START**.

Heap Management Routine

The **sbrk** function keeps track of the heap's growth and allocates space from the heap. It increments the heap pointer (__**HEAP**) by *size* bytes and returns the previous heap pointer value in register **D0**. If there is not enough space on the heap, **sbrk** returns -1.

System Functions

In addition to the functions provided in the Microtec C run-time library, the compiler assumes the existence of UNIX-style system functions. The source code to these system functions are provided with the libraries.

You must also modify the functions **sbrk** and **_exit** as well as the routines **inchrw**, **outchr**, **_START**, and **entry**. Descriptions of these routines are given in the section titled *User-Modified Routines*. The system functions are described in Chapter 5, *Library Extensions*, in this manual.

Sample versions of the system functions **inchrw**, **outchr**, **_START**, and **entry** are provided in the libraries.

Since embedded systems do not usually have a file system, the **close**, **lseek**, **open**, and **unlink** functions are provided with stubs only; they simply return 0. If **lseek**, **open**, or **unlink** are used, they will generate a link-time error indicating that the symbol **_WARNING_***xxx***_stub_used** is unresolved. The string *xxx* will be replaced by the function name. If the function is executed, the following message is written to **stderr**:

```
WARNING - xxx (stub routine) called
```

The **read** function, located in **s_read.c**, reads from the variable **_simulated_input**. **read**() returns the number of bytes read, 0 for **EOF**, or -1 to indicate an error. The **write** function, located in **s_write.c**, writes to the variable **_simulated_output** or

returns -1 to indicate an error. If you want to replace any of these functions with one of your own, link in your implementation before the run-time library.

The **creat** function uses the **open** function which is implemented as a stub. You must provide your own **open** function to allow **creat** to work in your environment.

The **sbrk** function is provided as a minimal implementation. If you are developing code for an environment that requires a different **sbrk**, you can provide a replacement for the **sbrk** function by linking in your implementation of **sbrk** before the run-time library. For example, an environment that might require a different **sbrk** might be one provided by a real-time executive.

Removing Unneeded I/O Support

You can reduce your code size by not linking in the full set of UNIX system functions included in the distribution, although you still may have a requirement for formatted I/O. In some embedded applications, the formatted I/O requirements are not extensive enough to involve a UNIX-style system. In other applications, the formatted I/O is only necessary during project development and will be removed before the code is finished.

In the above cases, you would probably want to have the library routine **printf** call your character output routine directly. Library routines such as **printf** implicitly write to **stdout** by passing characters from the library routines **putc** and **flsbuf** to the user-modifiable routine **write**. In a typical application, you would modify the source code provided for the **write** routine to work with the particular hardware in your system. If you do not link in the UNIX system functions, **write** is unavailable.

To allow formatted output to be sent directly to a low-level output routine, use one of the following two methods:

- Use **sprintf** rather than **printf** for formatted output. The **sprintf** function does the same formatting as **printf** but places the output in a user-specified string followed by the **NULL** character. The string may then be copied a character at a time to your output routine in a simple function such as:

  Example:

  ```
  str_out (char *str)
  {
        while (*str)
        _simulated_output = *str++;
  }
  ```

- Rewrite the **putc** routine so that all of the output characters will be available directly from **printf**. In the MCC68K library, the **printf** routine calls **putc** to output individual characters. Your **putc** routine will also intercept the

characters output by other library routines that use **putc**, such as **eprintf**, **fprintf**, **vfprintf**, and **vprintf**.

Example:

```
#include <stdio.h>
#undef putc /* shut off putc macro */
volatile extern char _simulated_output;
int putc(int c, FILE *stream)
{
        _simulated_output = c;
        return(c);
}
```

Note

The sample code provided here is not portable and relies on calls between library functions. This example may need to be modified for your version of MCC68K.

Removing Unneeded Floating-Point Support

If **printf**, **fprintf**, or **sprintf** is used and no floating-point values are ever read, you can avoid linking in full floating-point support by adding the following stub for **_ftoa** and compiling and linking it before you link in the library:

```
int _ftoa() { return 0; }
```

Alternatively, you could add the following stub for **_sfef** and compile and link it before you link in the library:

```
void _sfef() { return; }
```

In both cases, this technique reduces the size of the executable code.

Reserving a Register

Since it can be useful to reserve a register for shared data, system data, reentrant programs, or shared programs, the Microtec C compiler allows you to reserve **A***n* registers (from **A2**, **A3**, **A4**, **A5**, or **A6**) and **D***n* registers (from **D2**, **D3**, **D4**, **D5**, or **D6**). Up to three **A***n* and up to three **D***n* registers can be reserved during a given compilation. The compiler does not generate any code that uses the reserved register. If the **A6** register is reserved, **A5** will be used as the frame pointer.

A reserved register is usually used in conjunction with a Multitasking Environment (**MTE**). A multitasking environment handles one or more programs executing

simultaneously. A **MTE** can be thought of as an operating system; however, it might also be part of the application program or part of an embedded system.

To reserve a register, use the **-Kh***reg* option described under section *Produce Minor Code Generation Variations* in Chapter 2, *UNIX/DOS User's Guide*. Enter the option on the command line followed by the registers to be reserved. When this option is used, the generated code can be slightly less efficient in some cases.

Shared Program

When a register is reserved for a shared program, a single physical copy of the program can be shared by one or more users simultaneously. These can be single-tasking, multitasking, or interrupt-based programs, but only local variables and ROM variables (global or static) can be used. In this case, the reserved register is used as a pointer to the currently used data area (it points to only one data area at a time). Shared programs must be reentrant (see Figure 9-9).

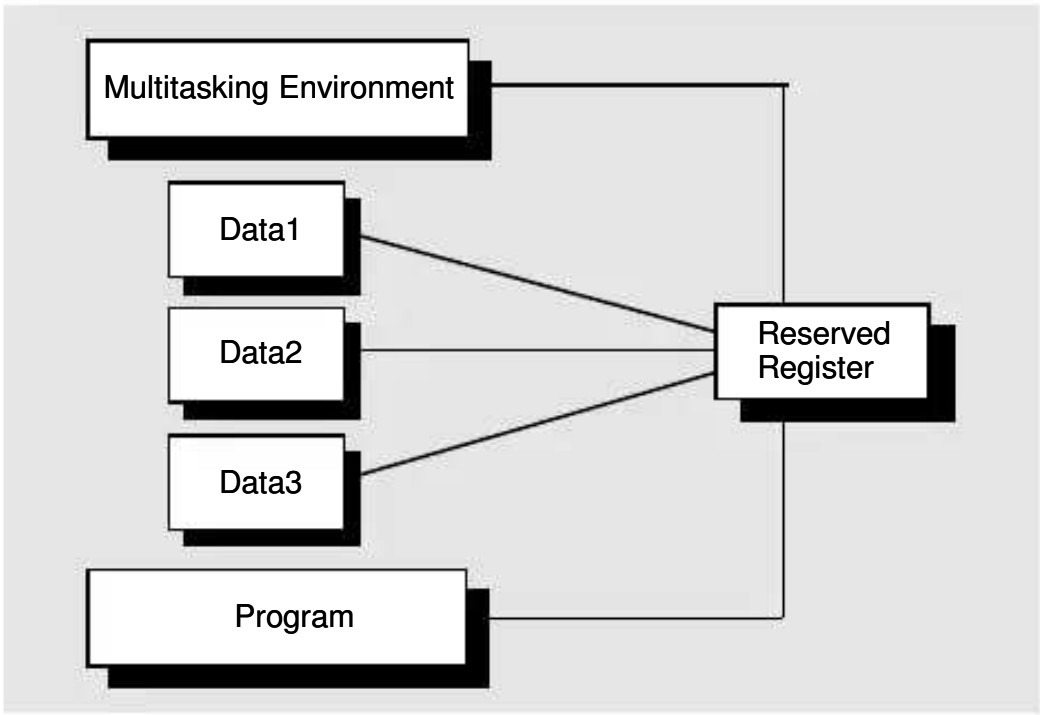

Figure 9-9. Shared Programs

Shared Data

When a register is reserved for shared data, a single physical copy of data in a multitasking environment can be shared among more than one program. The reserved register is used as a pointer to the shared data area as shown in Figure 9-10.

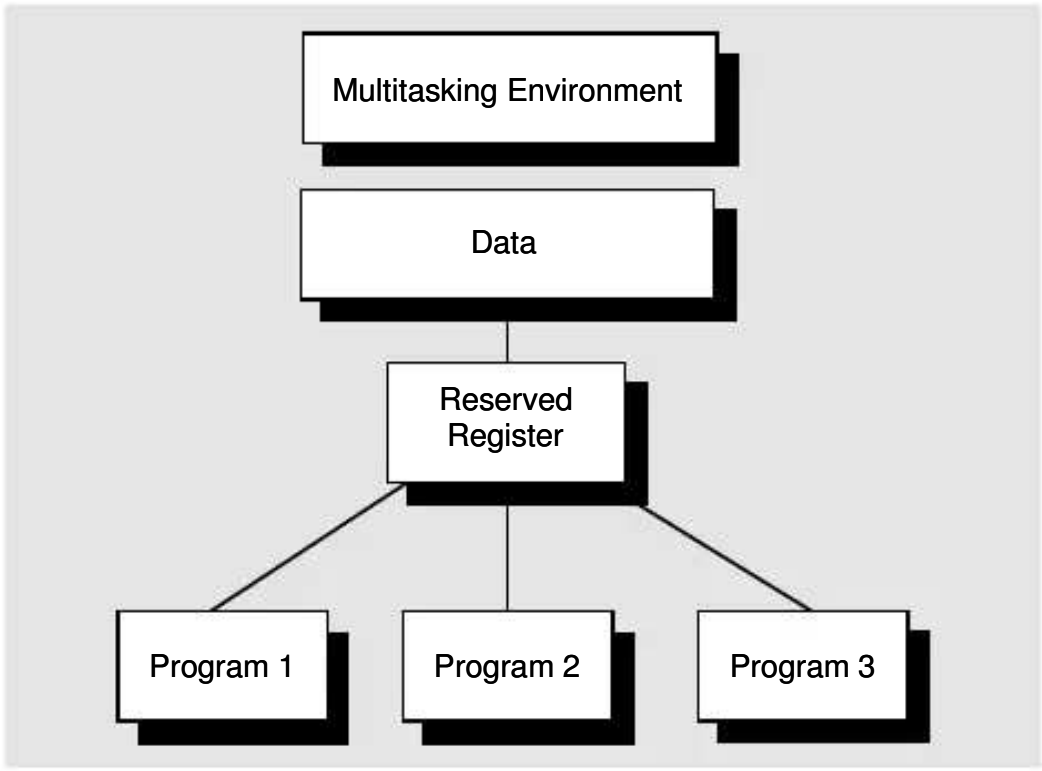

Figure 9-10. Shared Data

System Data

When a register is reserved to point to a system data area, special system data can be manipulated by a program. The multitasking environment sets up the system data area before the program is loaded. The multitasking environment must also load the address of the system data area into the reserved register. This configuration is shown in Figure 9-11.

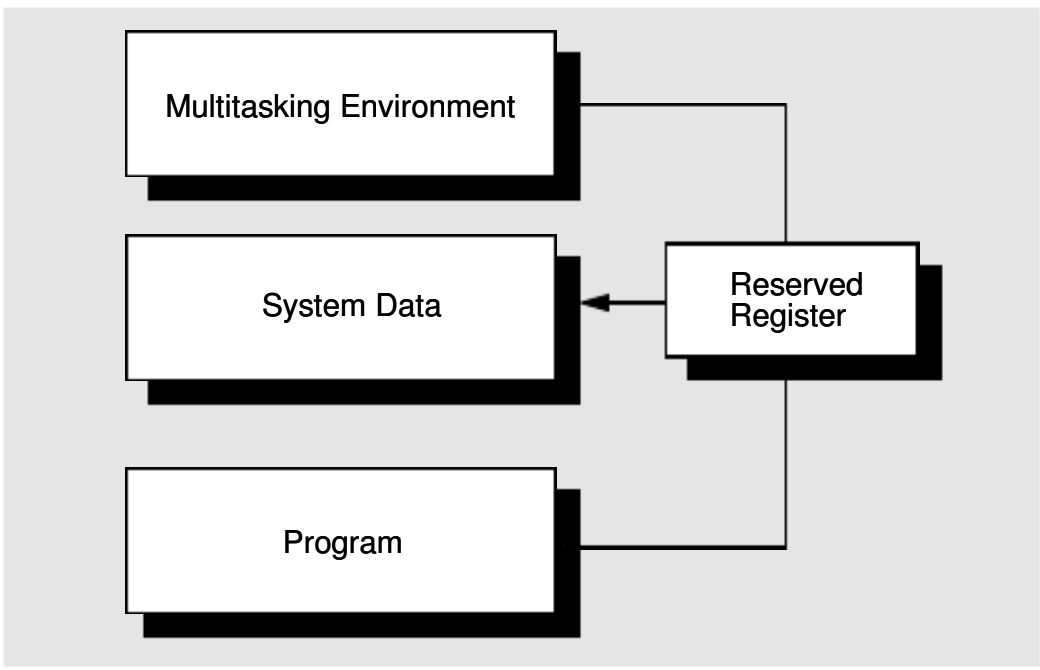

Figure 9-11. System Data

Special Purposes

It can be useful to reserve a register for special purposes. For example, a register can be used exclusively by an assembly language routine that is to be linked with C code. A register might also be reserved for a fast recall of a certain address or value, for locating data for ROM-based routines, and for memory management.

When a register is reserved for special purposes, it can either point to data or store the area that contains a special value. In Figure 9-12, a ROM-based routine reserves a register for a “hook” to find its data.

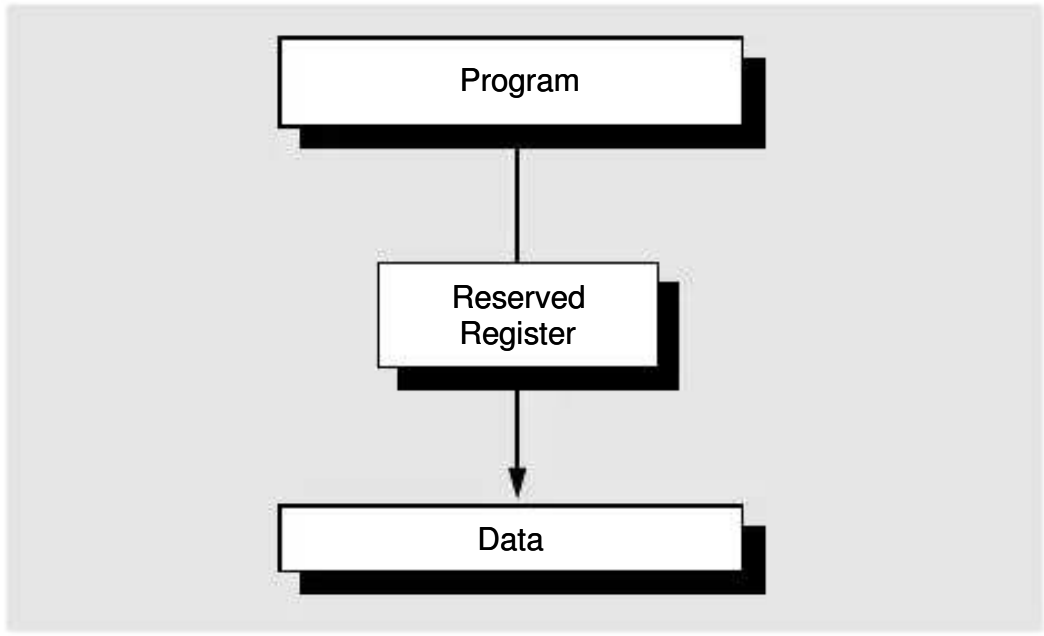

Figure 9-12. Reserved Register as a Pointer to Data

In Figure 9-13, the reserved register acts as a storage area containing a value used by the assembly language routine.

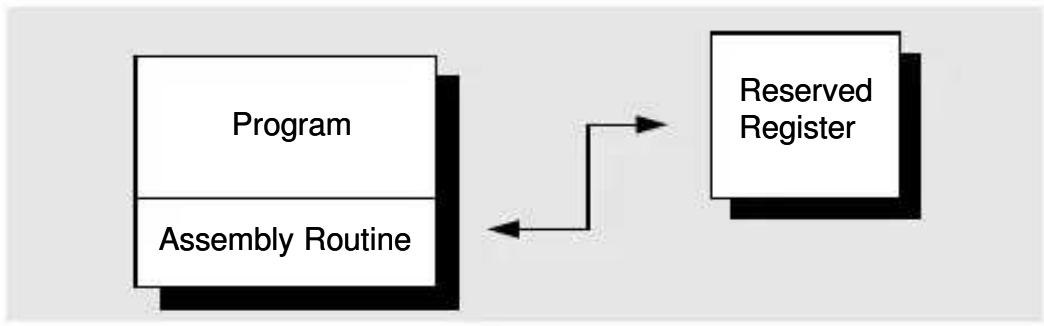

Figure 9-13. Reserved Register as a Storage Area

Data Initialization

Variable values can be initialized at run time or compile time.

Example:

Run-Time Initialization **Compile-Time Initialization**

```
int i;                          int i = 15;
i = 15;
```


Code is always ROMable when variables are initialized at run time. In this example, the constant value `15` is stored in the **code** section.

When a variable is initialized at compile time, the compiler allocates the variable, along with its value, in the **vars** section. Often the **vars** section will be located in RAM. For most ROM-based embedded systems, this presents a problem. When your program starts executing, RAM is typically not initialized with the variable values contained in the **data** section. The variable needs to be in ROM.

The run-time libraries provided with the MCC68K Compiler contain initialized variables. So, the problem exists for run-time library initialized variables, as well as for user-initialized variables.

Saving Initialized Variables in ROM

Because RAM is typically not initialized with the expected variable values when your program starts executing, many embedded programmers use the convention that only constants can be initialized on the variable declaration. The value of the variables cannot be changed at run time. In this case, the **vars** section can be placed in ROM, along with the **code** section.

If you do not want variables initialized at compile time on the data declaration to be constants, you can use the **INITDATA** facility in most ROM/RAM systems. This facility initializes the values of these variables when the program restarts.

Using the Linker

This section provides linker command examples, including an example using a ROM-based system. See the section Chapter 8, *Run-Time Organization*, in this manual for a description of the default section names the linker uses.

Default Sections

Refer to Chapter 8, *Run-Time Organization*, in this manual for information about default linker sections. Refer to the section *Choose Address Mode for Sections* in Chapter 2, *UNIX/DOS User's Guide*, for information on locating and addressing the various code and data sections.

Libraries

For information about which libraries to use, refer to the section *Libraries* in Chapter 2, *UNIX/DOS User's Guide*.

Messenger Symbols

A messenger symbol is a compiler-generated symbol that is resolved by the run-time library. This symbol is identified by four preceding underscores.

The **-f** compiler option generates the messenger symbol **____FPU**. If this messenger symbol is present, the XRAY Debugger sets the **@fpu** pseudo-register to 1 (indicating the presence of the 68881 floating-point coprocessor).

INITDATA Command

The LNK68K linker command **INITDATA** provides a method for initializing program variables in RAM at program start-up time. If the **INITDATA** command is used, the linker will create the section **??INITDATA**. This section will contain the initialization values for the section(s) specified with the **INITDATA** command. At program start-up time, the **initcopy** function will copy the initialization values from the **??INITDATA** section to the section(s) specified with the **INITDATA** command.

The Microtec run-time libraries contain the **initcopy** function. By default, the **initcopy** function is called from the **START** function in **csys.c**. The **csys.c** file on the distribution includes a call to **initcopy**.

If you wish to disable the **INITDATA** command, follow these steps:

1. Edit your linker command file and remove this command:

   ```
   initdata vars
   ```

2. Do one of the following:

   a. Define the preprocessor macro **EXCLUDE_INITDATA** either on the compiler command line with the **-D** option or in your program with the following directive:

      ```
      #define EXCLUDE_INITDATA 1
      ```

   b. Edit the file **csys.c** and remove the **#if EXCLUDE_INITDATA** preprocessor directives and their associated **#endif** directives. Then recompile the file and link it in before the library.

For more information on the **INITDATA** command, see the *Assembler/Linker/Librarian User's Guide and Reference for the 68000 Family.*

Linker Command Example

The following linker command file creates the memory configuration given in Figure 9-14.

```
listabs publics, internals
listmap publics
format ieee             ; Output file format
extern ENTRY            ; Force load of initialization
                        ; routine
sectsize heap=$8000     ; Set size of heap
sectsize stack=$1000    ; Set size of stack
common stack=$F000      ; Set start of stack section

order stack             ; Stack section
order literals,strings,const,code   ; ROM Section
order vars,zerovars,ioports,heap    ; RAM Section
load sieve              ; User application routine
load mcc68kab.lib       ; C run-time library
end                     ; End of command file
```

The compiler places code and data in the sections described previously in the section *Default Sections* in this chapter. The example assembly module **ENTRY** uses section **heap** for the heap.

By default, the linker begins assigning memory at address 0, which is unsuitable for many real applications because addresses 0 to 3FF are used as interrupt vectors and space needs to be allocated for the stack. Use the **COMMON** command to specify the starting point address of the **stack** section.

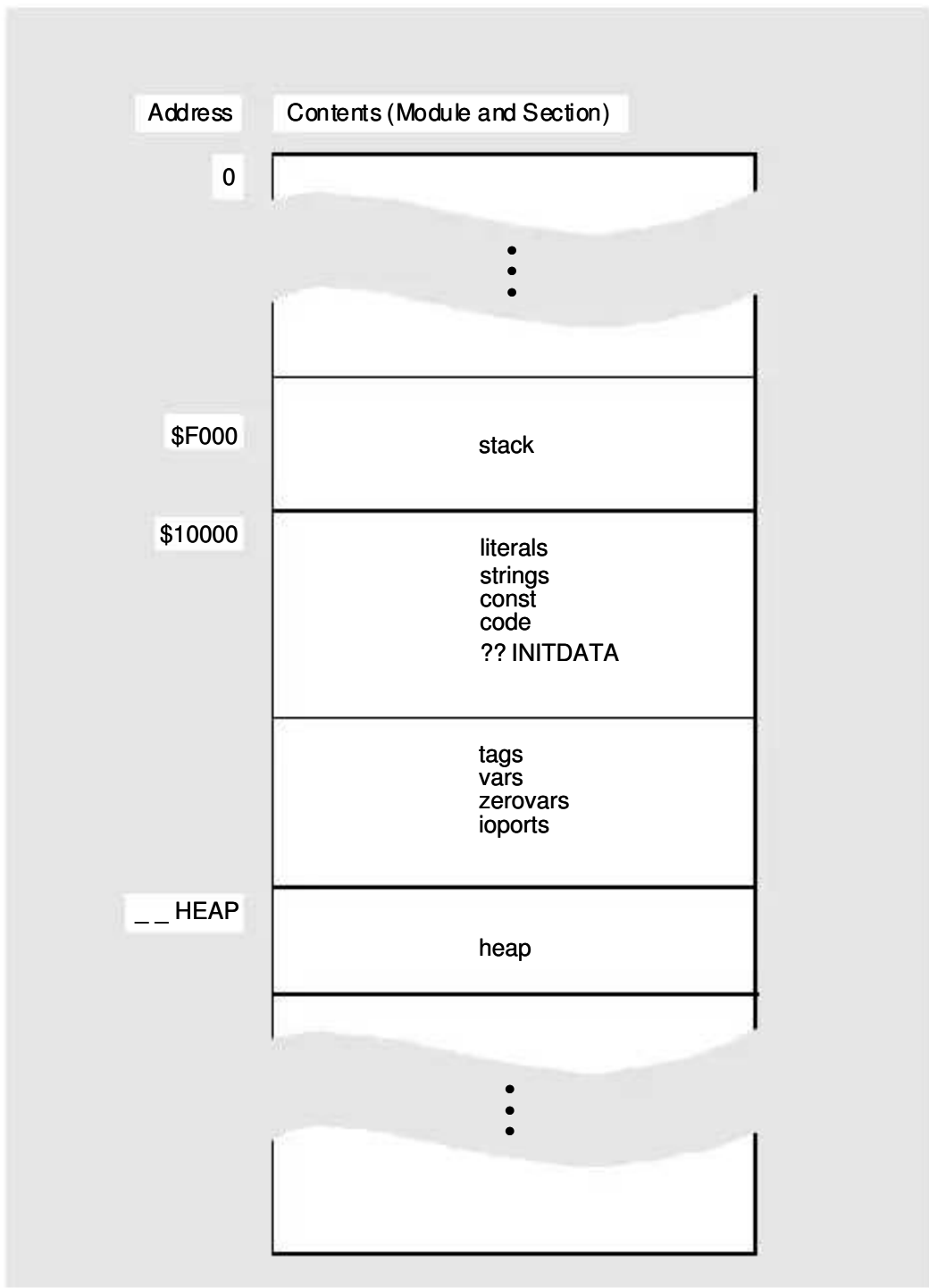
Address
Contents (Module and Section)
0
$F000
stack
$10000
literals
strings
const
code
?? INITDATA
tags
vars
zerovars
ioports
_ _ HEAP
heap

Figure 9-14. Memory Configuration

Linker Command Example (ROM-Based System)

In some applications, such as ROM-based dedicated systems, code, data, and stack sections are to be assigned to preallocated areas of the processor's address space. In the following linker command file, it is assumed that the **literals** section is to be placed in the ROM area starting at F000, the **zerovars** section is to start at location 2000, the **vars** section is to start at location A000, and the stack is to reside in the 0 to 2000 region.

```
listabs publics, internals
listmap publics                 ; Linker options
format ieee                     ; Output file format
extern ENTRY                    ; Force load of
                                ; initialization routine

sect ??INITDATA=$E000
sect literals=$F000             ; Code section address
sect zerovars=$2000             ; Uninitialized data
                                ; section address
sect vars=$A000                 ; Initialized data
                                ; section address
common stack=$1000              ; Set start of stack section
sectsize heap=$8000             ; Set size of heap
sectsize stack=$1000            ; Set size of stack
;
order stack                     ; Stack section
order  zerovars,ioports,vars,??INITDATA
order  literals,strings,const,code,heap
initdata vars                   ; Put init values in ??INITDATA
load entry                      ; Main entry point
load sieve                      ; User-application routine
load mcc68kab.lib               ; C run-time library
end                             ; End of command file
```

Figure 9-15 shows this memory configuration.

The initialized variables will be allocated in section **vars** and their initialization values will be allocated in the **??INITDATA** section, which will be placed at E000. To copy the initialization values from **??INITDATA** (ROM) to **vars** (RAM), **initcopy** should be called from the **entry** function.

Using the allocation map generated by the linker, further adjustments to the starting addresses might be necessary to ensure sufficient space for each program section.

The ABSOLUTE linker command can be used to control what sections are placed in the output file. For more information about this and other linker commands, see

the *Assembler/Linker/Librarian User's Guide and Reference.*

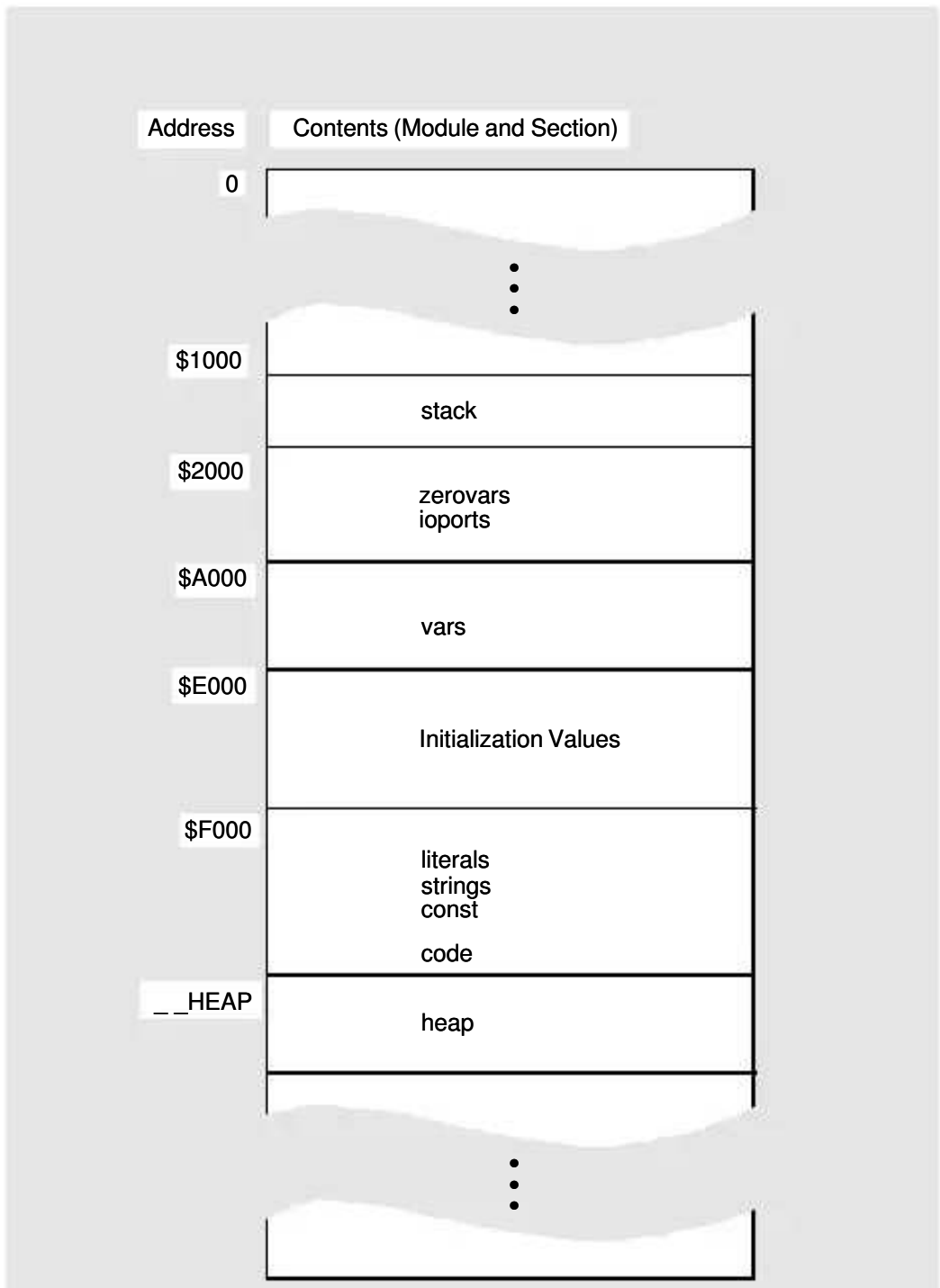

Figure 9-15. Memory Configuration (ROM Based)

Interrupt Handlers

Interrupt handlers perform the necessary functions when an interrupt occurs. They preserve the values in various registers and storage locations and transfer control to routines to service the interrupt.

By default, the Microtec C compiler assumes that the values in registers **A0**, **A1**, **D0**, and **D1** may be destroyed by each called function. Therefore, you may want to declare a function as an interrupt handler to force the values of these registers to be saved on the stack on entry to the function and to be restored on exit.

A function can be declared as an interrupt handler by using the **interrupt** keyword. Interrupt functions should have no parameters and should return the **void** type.

Example:

```
interrupt void handler (void)
{
...
}
```

By default, a function declared as an interrupt handler will return to the calling function with the **Return To Exception** (**RTE**) instruction rather than the standard **RTS** instruction.

If the **-Kr** option is used, the interrupt handler will return with an **RTS** instruction.

Optimizations 10

Overview

Optimizations are techniques employed by compilers to improve the object code produced. The improvements include reduced program code size and increased execution speed. The extent of improvement depends on the content of an individual program, its coding style, and the compiler's ability to recognize and optimize certain constructs.

The Microtec C compiler performs many commonly known optimizations, which are described in this chapter. Some of these optimizations are applied before, some during, and some after generating code for a C expression. Most optimizations are independent of the target processor, but some are specific to the microprocessor and the calling conventions.

In general, for a given C expression, the compiler tries to generate optimal code that is slightly in favor of code size. Sometimes the compiler finds an alternate code sequence that results in significant improvement in code size but also in a substantial reduction in execution speed or vice versa. Under such circumstances, you have a choice of optimizing in favor of speed or code size with the **-Ot** and **-Os** command line options. Refer to the *Optimize Code* in Chapter 2, *UNIX/DOS User's Guide*, for details on these options.

General Optimizations

The Microtec C compiler performs several optimizations before generating code for a C program. These optimizations usually involve constant expressions or expressions with known results that can be evaluated at compile time. Programmers seldom write explicit expressions with only constants, but it is not uncommon for them to write an expression with defined constants.

Algebraic Simplification

These optimizations replace an expression with an equivalent but simplified expression.

Examples:

```
a * 1   ⟶   a
b - 0   ⟶   b
c == c  ⟶   1
x -= x  ⟶   x = 0
```

Redundant Code Elimination

Code sequences that produce no real side effect are considered redundant and can be eliminated. They usually evolve as the result of other optimizations. If the variable is declared as **volatile**, this optimization is not performed.

Examples:

```
a = a + 0;  ⟶   No code generated
b + c;      ⟶   No code generated
x &= x;     ⟶   No code generated
```

Strength Reduction

These optimizations replace operations with equivalent but less expensive (reduced strength) operations.

Examples:

```
a * 128             ⟶   a << 7
(unsigned) b / 16   ⟶   b >> 4
(unsigned) c % 32   ⟶   c & 31
```

Global Optimizations

The Microtec C compiler performs several optimizations that make programs smaller and faster by applying global data flow and control flow analysis on the entire function.

Dead Code Elimination

This optimization eliminates unreachable code (i.e., “dead code”). Unreachable code occurs through user error or conditional compilation techniques that exploit the preprocessor. Unreachable code sequences are usually detected after performing

jump optimizations. Unlabeled instructions immediately following an unconditional jump can be removed. Conditional expressions that can be reduced to constants also produce unreachable code.

Examples:

```
#define TRUE 1
#define FALSE 0
#define DEBUG 0
```

```
while (a)
{
      test1();
      if (b)
            continue;
      else
            break;
      test2();                    ⟶  Dead code
}
if(DEBUG)                         ⟶  Dead code
      printf("debug1 \n");        ⟶  Dead code

while (TRUE)
{
      test1();
      test2();
}                                 ⟶  Dead code
return (b);                       ⟶  Dead code
```

```
      goto 11;
      i1 = 3;
      i2 = 4;        ⟶  No code generated
      i3 = 5;
11:
      i4 = 3;

a  =  ( 3 > 0 ) ? xx (6) : yy (7);  ⟶  a=xx(6);
if ( a != a ) b = 5;                ⟶  No code generated
```

If a breakpoint is set on the line containing the statement `i2=4` in this example, the actual breakpoint will be set on the line containing the statement `i4=3`. Then, whenever a jump is made to `11`, the actual breakpoint will stop the program and will cause confusion.

A more typical example of dead code removal applies when the compiler determines that the condition of an **if** statement equates to a constant, which occurs if every operand is a constant or a variable has been assigned to a constant value.

Example:

```
#define control 0
int auxcontrol;
. . .
auxcontrol = 0;
if (control | auxcontrol)  >>> These lines
{                          >>> are removed
  i1 = 3;                  >>>
  i2 = 4;                  >>>
}                          >>>
else                       >>>
  i3 = 5;
```

The preprocessor symbol `control` is replaced by the text `0` and is interpreted as the constant zero. The compiler determines that the integer variable `auxcontrol` still retains the value of zero at the beginning of the `if` statement and uses zero for `auxcontrol`. The compiler evaluates the `if` expression `(control | auxcontrol)` as `(0 | 0)` or zero and causes the first part of the `if` statement to become dead code and to be removed.

Factorization

Factorization optimization is particularly useful for storing a frequently used static address in a register. It references the static value or address through register indirect addressing and stores it at the beginning of the function. Factorization optimization is applied to the entire function.

Example:

```
test(1);movea.l#_test,a2
test(2);pea 1
test(3);jsr (a2)
      pea   2
      jsr   (a2)
      pea   3
      jsr   (a2)
```

Global Constant Propagation

When a constant is assigned to a variable, the constant may be carried forward to the subsequent uses of the value of that variable until a new value is assigned to that

variable. This optimization is performed across basic blocks.

Example:

```
;  var1 = 2;
       moveq       #2,d0
       move.l      d0,_var1     /* var1 is global variable. */
;  if (var1)                    /* Will not do test, var1 */
                                /* is known to be 2. */
;      i = var1;                /* 2 will be used instead */
                                /* of var1. */
       moveq       #2,d1
;  f(var1,i);                   /* 2 will be used instead */
                                /* of var1. */
       move.l      d1,-(sp)
       pea         2
       jsr         _f
       addq.l      #8,sp
```

Global Copy Propagation

When a variable that is "cheap" to access (e.g., a register variable) is assigned to a variable that is "expensive" to access (e.g., a global variable), the cheap variable may be carried forward to the subsequent uses of the expensive variable until a new value is assigned to the expensive variable. This optimization is performed across basic blocks.

Example:

```
;  var1 = i;                    /* i is in d2. */
       move.l      d2,_var1     /* var1 is global variable. */
;  if (var1)                    /* Will test i instead */
                                /* of var1. */
       tst.1       d2
       beq.s       L2
;      f(var1+2);               /* Function will pass i+2 */
                                /* instead of var1+2. */
       move.l      d2,d0
       addq.l      #2,d0
       move.l      d0,-(sp)
       jsr         _f
       addq.l      #4,sp
  L2:
```

Global Value Propagation

If global optimization (**-Og**) is selected, the values of variables as well as constants are remembered. Once the value of a global variable is placed into a register, future references will use the register value instead of re-accessing the global variable.

Example:

```
glob1 = 2;

i = glob1;        /* The register value will be used */
                  /* instead of globl */

f(glob1,i);       /* if -Og, the register will be used
                  /* instead of globl */
```

Register Allocation

Optimizing register usage throughout an entire routine is also referred to as register coloring. It is particularly valuable for keeping most of the local variables in the registers at all times. Using data flow analysis, the compiler finds the lifetime of each variable. The register coloring algorithm can then increase the number of variables that are stored in registers by using the same register for several variables in the same routine.

Unused Definition Elimination

An unused definition is an assignment in which the left-hand side variable is either never used or used after being reassigned. The compiler eliminates unused definitions.

Example:

```
. . .
. . .
i = j + 1;  /* if the value of the variable i is never used

. . .          again, this statement will be eliminated */
. . .
```

Example:

```
i = j + 1;  /* this assignment will be eliminated since i
               will not be used before it is reassigned */
i = k + 1;
```

Loop Optimizations

This section lists optimizations that can be applied to loop constructs.

Array Operator Synthesis

Because the C language does not have an array assignment operator, **for** loops are often used to assign a value to all elements of an array or to assign one array to another.

Example:

```
char arr[HI_BND], arr2[HI_BND];
for (i=0; i<HI_BND; i++)
     arr1[i] = 'a';
for (i=0; i<HI_BND; i++)
     arr2[i] = arr1[i];
```

Without this optimization, constructs like those shown in this example would result in very inefficient code because of the number of loop iterations required.

The Microtec C compiler recognizes and optimizes constructs of the following form:

```
for (i=const1; i<const2; i++)
     arr[i]=const3;
for (i=const1; i<const2; i++)
     arr[i]=arr2[i];
```

Depending on the number of elements to be assigned, the compiler may "unroll" the **for** loop or may minimize the number of iterations required. For example, to initialize a character array, the compiler will use **move.l** rather than **move.b**, thereby requiring only twenty-five percent as many iterations.

Stepping through a **for** loop will not work as expected with this type of optimization. The **for** loop is considered to be a single statement. One XRAY Debugger **STEP** command will be sufficient to step over the entire **for** loop.

Because this optimization may result in larger (but faster code), it can be disabled by a compiler option. Refer to section *Optimize Code* in Chapter 2, *UNIX/DOS User's Guide*, for information about the appropriate option.

Loop Invariant Code Optimization

Loop invariant analysis speeds up loops by eliminating computations of invariant expressions and addresses. These computations are moved out of the loop and the value is stored in a register. This optimization is valuable for removing array subscripting from a loop when the subscript is a variable or expression that is not

modified in the loop.

The loop control code generated by an **if** or other similar construct will be removed at compile time if the compiler determines that the controlling loop variables will

cause a loop to be executed only once. The entire loop is removed similarly to dead code if the compiler determines that the loop control variables keep the loop from ever being executed.

Example:

```
for (i1=1; i1<=1; i1++)
{
      ival=ival*2+1;
}
```

→

```
ival=ival*2+1
```

The loop invariant code optimization searches loops for expressions that yield the same result regardless of the number of times the loop is executed. The optimization then places the expression before the loop.

Example:

```
while (i<10)                    move.l      d3,d2
{                               add.l       d3,d2
      test(j+k+l);              add.l       d3,d2
      i++;                      bra.s       L2
}                   L1:
                                move.l      d2,-(sp)
                                jsr         _test
                                addq.l      #4,sp
                                addq.l      #1,d3
                    L2:
                                moveq       #10,d0
                                cmp.l       d3,d0
                                bgt.s       L1
```

Loop Rotation

In both the **for** and **while** loops, the controlling expression is located syntactically at the top of each loop. When translating these loops, the controlling expression is rotated to the bottom. This rotation reduces the number of instructions within the body of the loop.

The **while** loop on the left is translated as if it were written as the sequence of statements on the right:

```
while (expression)          goto test;[1]
{                           loop:
statements;                 {
}                           statements;
                            }
                            test: if (expression) goto loop;
```

The **for** loop on the left is translated as if it were written as the sequence of statements on the right.

```
for (expr1; expr2; expr3)   expr1;
{                           goto test;[2]
statements;                 loop:
}                           {
                            statements;
                            }
                            expr3;
                            test: if (expr2) goto loop;
```

Strength Reduction and Index Simplification

Strength reduction and index simplification is also referred to as induction variable elimination. The values of induction variables increment or decrement by a constant value for each iteration of the loop. These variables are often used to count or index an array. Also, multiples of these variables can be computed. It is often possible to replace all but one through the process of induction variable elimination.

Example:

```
for (i=1; i<10; i++)moveq#1,d1
     ary[i] = i; lea.l -76(a6),a0
           L1:
           move.ld1,(a0)+
           addq.l#1,d1
           moveq #10,d0
           cmp.l d1,d0
           bgt.s L1
```

1. If the compiler determines that *expression* is initially true, then this `goto` statement is suppressed.
2. If the compiler determines that *expr2* is initially true, then this `goto` statement is suppressed.

Local Optimizations

Local optimizations are generally applied to small sections of the generated code. Some of the jump optimizations described in the *Jump Optimizations* section in this chapter can also be considered local optimizations.

Common Subexpression Elimination

Common subexpression optimizations remove expressions that have been previously evaluated.

Example:

```
if (a+b > 3)                move.ld2,d0
    test(a+b);              add.l d3,d0
                            move.ld0,d2
                            moveq #3,d1
                            cmp.l d0,d1
                            bge.s L1
                            move.ld2,-(sp)
                            jsr   _test
```

Constant Folding

Arithmetic and logical operations involving only constants are evaluated at compile time.

Examples:

```
i = 1 + 2 * 3 + 4       ⟶   i = 11
(a  >>  3)  >>  1       ⟶   a >> 4
i + 5 + j - 6 + k - 7   ⟶   i + j + k - 8
(4 & 1) || (5 > 7)      ⟶   0
```

Generating Code for switch Statements

The Microtec C compiler applies a special optimization to generate efficient code for the **switch** statement. Depending on the number of cases and the range of case values, the compiler uses one of the following methods to generate a code sequence for the **switch** statement:

- If there are three cases or less or the jump table is sparse, a series of com-

pare and jump instructions is generated. The order of comparing the cases is the same as they appear in the program. Therefore, you can improve the execution speed by placing the most frequently occurring case first.

- If the table is fairly dense (i.e., not too many holes), the compiler uses a jump table indexed by the switch value. The compiler generates a short sequence of instructions to scale the switch value and to compare to the size of the jump table before indexing. This method has the fastest execution speed of the three methods.
- If the jump table in the previous method is sparse, two parallel tables are constructed instead: one for the case values and one for the jump labels. The compiler generates a short sequence of instructions to linearly search the case value table for the switch value. If it finds the switch value, the table entry offset is used to index into the jump label table. The order of case values is the same as they appear in the program, depending on their range.

The compiler estimates the code and table size required for each method described and then selects the one with the minimum code size. When optimizing for time, the compiler will favor the second method (i.e., dense table).

Redundant Load and Store Elimination

Loading the same variable after storing it is redundant and can be eliminated. This optimization can also be applied to pushing the same variable onto the stack.

Example:

```
i = expression;      (code for expression)
j = i & 5;           move.l   d0,_i
                     move.l   _i,d0      Instruction eliminated
                     andi.l   #5,d0
                     move.l   d0,_j
i = expression;      (code for expression)
printf ("%d", i);    move.l   d0,_i
                     move.l _i,-(sp)          move.l d0,-(sp)
```

Jump Optimizations

The Microtec C compiler performs jump optimizations after generating code for each function. The compiler examines every jump instruction and its target location to recognize specific patterns for optimization. Since most jump optimizations create opportunities for further improvements, the optimizations are applied iteratively until no further changes result.

tively until no further changes result.

Branch Tail Merging

The compiler combines common instructions preceding two jump instructions that jump to the same location. This optimization is enabled with the **-Og** option.

Example:

```
jsr   _func1 jsr  _func1
addq.l #1,d0L12:  addq.l #1,d0
add.l d0,8(a6)   add.l d0,8(a6)
bra.s L1     bra.s L1
...          ...

jsr   _func2 jsr  _func2
addq.l #1,d0bra.s L12
add.l d0,8(a6)
bra.s L1
...
```

In this example, the three instructions following the jump instruction on the left were combined under the `L12` label.

Example:

```
1     if(i)
2     {
3           i2 = 3;
4           i1 = 7;
5     }
6     else

7     {
8           i2 = 21;
9           i1 = 7;
10    }
```

In this example, the compiler will not generate code for two `i1=7` statements. Instead, it will generate code for the `i1=7` statement on line `9` and only a jump statement for the `i1=7` statement on line `4`.

Code Hoisting

In a similar manner to branch tail merging, code hoisting optimization moves

common code up in the program to a common point.

Example:

```
cmp.l 12(ab),d0                    cmp.l 12(ab),d0
    bge.s L3                  ⟶       lea.l (a1,d1.l),a0
    lea.l (a1,d1.l),a0                bge.s L3
...                                              ...
L3: lea.l (a1,d1.l),a0          L3:
...                                    ...
```

This optimization may be turned off with the appropriate compiler command line option. Refer to section *Optimize Code* in Chapter 2, *UNIX/DOS User's Guide*, for more information.

Example:

```
k = i+prime;
while (k<10)
{
     flags[k] = 0;
     k += prime;
}
```

The instruction `k += prime` at the bottom of the loop is combined with the instruction `k = i+prime` before the loop entry point. When stepping through this loop in the debugger, the program counter will step out of the loop and highlight the `k = i+prime` statement. If you examine the resulting assembly code, you can see that only part of this statement is being executed (`k=`*temporary_value*).

Cross-Jump Optimization

The compiler replaces a jump and some prior instructions by a jump to an earlier location that has the same instructions.

Example:

```
     move.l    d0,-(sp)
     jsr       _func
     bra.s     L1          ⟶        bra.s L12
     ...                            ...
     move.l    d0,-(sp)    L12:  move.l d0,-(sp)
     jsr   _func                    jsr   _func
L1:  ...                     L1:   ...
```

This optimization may be turned off with the appropriate compiler command line

option. Refer to section *Optimize Code* in Chapter 2, *UNIX/DOS User's Guide*, for more information.

This optimization can change breakpoints and perceived program flow and can affect the values of variables.

Multiple Jump Optimization

Generated code for complex control flow statements contain jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These multiple jumps can be rerouted to jump directly to the target locations.

Example:

```
        bra.s L1                        bra.s L2
        ...                             ...
        bne.s L1          ⟶             bne.s L2
        ...     .                       ..
L1:     bra.s L2                L1:     bra.s L2
        ...                             ...
```

This optimization changes the perceived program flow.

Redundant Jump Elimination

A jump instruction that jumps to a location that immediately follows it can be eliminated. Redundant jumps usually appear after applying other jump optimizations.

Example:

```
        bra.s L3      This instruction will be eliminated.
L3:
```

Example:

```
         if (i3==3) goto L1;     >>> if(i3==3) goto L2;
        il = 3;
        i2 = 3;
L1:
        goto L2;
        k = 5;
L2:
```

In this example, the `if` statement will be changed as shown. A breakpoint at label `L1` will not be encountered when `i3==3`.

The next example shows a more typical example of redundant jumps.

Example:

```
1     while (x)
2     {
3           array(x) = 0;
4           x = x+1;
5           while (i)
6           {
7                 istat(i) = 0;
8                 i = i-1;
9           }
10    }
```

Short/Long Displacement Optimization

During code generation, the compiler does not know the exact displacement between a jump instruction and its target location. Many of these jump displacements are changed as a result of applying jump optimizations. As a last step in jump optimizations, the compiler determines the exact displacements and replaces them with the shortest instructions possible for conditional and unconditional jumps.

Example:

```
      cmpi.l    #20,d0                    cmpi.l #20,d0
      bne.l     L1                        beq.l L2
      bra.l     L2          ──►     L1:   ...
L1:   ...                                 bra.s L3
      bra.s     L3
```

Function In-Lining

In-line expansion of a user function saves the overhead usually associated with function calls, parameter passing, register saving, stack adjustment, and value return. Sometimes, further optimizations become possible because the actual parameters used in a call become visible, and the side effects of function calls also become exposed to the caller.

Example:

```
func(char op, int a, int b)
{
      if (op == 1)
            return(a+b);
      else
            return(a-b);
}

main()
{
int x, y, z;
...
                  ; Parameter passing of 1, x, y removed.
                  ; Register saving in func removed.
                  ; if (op == 1), comparison removed.
                  ; Register restoring in func removed.
                  ; Return value removed.
                  ; Stack adjustment after call removed.
                  ; The code generated is:

      z = func(1, x, y);      /*    move.l      d3,d4 */
                              /*    add.l       d2,d4 */
}
```

inline Keyword

Any function may be qualified as in-line to tell the compiler which functions are the best candidates for in-lining when invoked within the same module. A function will not be in-line under the following circumstances:

- The inline optimization (**-Oi** option) has not been enabled.
- The Microtec extensions (**-x** option) have been disabled. The **inline** keyword will not be recognized.
- The function designated for in-lining is not a leaf function (a leaf function is a function that does not contain invocations of other functions).
- The global optimizer (**-Og**) has not been invoked.
- The favor space optimization (**-Og** and **-Os**) has been selected.
- The function declares static local variables.

- The function is deemed too complex or too large by the compiler.
- The invocation point is not favorable. For example, not enough registers are available to make in-lining convenient at a given point in the program.

- The function is an **extern** function. As in-lining is not performed across modules, specifying **inline** for an **extern** function will have no affect.

Syntactically, the keyword **inline** is part of a declaration specifier and can be placed exactly as the **keyword interrupt**.

Instruction Scheduling

Improved performance can be achieved by rearranging the instruction sequence to avoid stalling the instruction unit pipeline. This includes overlapping CPU instructions with FPU instructions and inserting useful instructions between those instructions that could cause pipeline stall because of data dependence.

Machine-Dependent Optimizations

The Microtec C compiler also performs optimizations that are specific to the calling convention or architecture of the target machine. For information on compiler command line options for optimization, refer to section *Optimize Code* in Chapter 2, *UNIX/DOS User's Guide*.

Generating Code for Function Prologue and Epilogue

The standard code sequences for function prologues and epilogues are described in Chapter 7, *C Calling Conventions*, in this manual. The Microtec C compiler generates the full prologue or epilogue only when it is absolutely necessary or when the **-Kf** option is used.

A prologue will not be generated if a stack frame is not needed, if there are no local variables, or if the function does not return a structure.

If only one register needs to be saved, the compiler will use the **move.l** instruction instead of the **movem.l** instruction. This saves 6 cycles.

In-Line Library Function Expansion

Certain library functions can optionally be replaced with in-line code. In addition to removing the overhead of a function call, this can also enable leaf and scheduler optimizations to be performed.

Example:

```
char stg[100]
f3()
{
      strcpy(stg, "constant string");
}
```


Grouping Stack Adjust Instructions

The calling convention requires the calling routine (the caller) to remove the function arguments from the stack after the call. The stack pointer is adjusted by adding a constant that equals the total size of the function arguments pushed. The compiler tries to combine two or more stack adjust instructions by accumulating the adjustment to the last instruction.

Example:

```
/* code with -Oc enabled */
jsr   _function1
jsr   _function2
lea.l 12(sp),sp

/* code with -nOc enabled */
jsr   _function1
addq.l #8,sp
jsr   _function2
addq.l #4,sp
```

This optimization cannot be applied if the first call is one of the function arguments of the second call or if any jump or label occurs between the two function calls. The compiler accumulates a maximum of 50 bytes to reduce the stack growth. This maximum is particularly important for recursive function calls.

You can disable this optimization by using the **-nOc** option.

Indexing Arrays

When indexing into arrays that are less than or equal to 64K bytes, the index expression will be computed in **short** rather than **int**. This will result in faster but not smaller code.

This optimization will not be applied to arrays of unknown size (which can be declared as **[ ]**), arrays of size 1 (sometimes used to denote an unknown size), or arrays of greater than 64K bytes.

Char Operations Where Possible

When comparing masked 16-bit expressions with 8-bit values or when assigning 16-bit expressions to 8-bit results, 8-bit arithmetic is used when the result is not affected.

affected.

Example:

```
char c;
int i,j;
/* 16-bit operations required (int result) */
j=i+2;

/* 8-bit operations allowed (char result) */
c=i+2;

/* 8-bit shortcut for masked int */
if ((i & 255) == 27)
```

Compiler Output Files 11

Assembler Source File

The MCC68K Compiler produces an assembler source file that is in Microtec and Motorola compatible format. There are several assembler directives that can optionally pass variable name and data type information from the compiler directly to the XRAY Debugger.

Advantages of Producing an Assembly File

You can examine and optimize the generated assembly code file by hand. Be careful when you modify the assembly code; an incorrect change can produce unpredictable results.

A compiler command line option is available to intermix the high-level source statements as comments with the generated assembly code in the output file. See the section *Specify Format of Output Files* in Chapter 2, *UNIX/DOS User's Guide*.

C variable and line numbers are mapped into symbol names and placed in the output object module as described in the following sections.

Variable Names

The Microtec C compiler alters the names of external, public, and static variables to avoid possible conflicts. These naming changes are described in the following sections.

External and Public Variable Names

The Microtec C compiler alters the names of all variables and functions declared in the C input source. Variables and functions are prefixed with a leading underscore to avoid possible conflicts with assembler reserved words. The options described in the section *Modify Naming Conventions* in Chapter 2, *UNIX/DOS User's Guide*, can change this behavior. These options are not recommended because they prevent access to the standard C library functions (all library functions have a leading underscore).

For a C program to access an assembly language routine or data item, the item name must begin with an underscore and be declared as public in the assembly routine. The C program should refer to the assembly language item without the underscore.

The following example shows function naming and global data in C and assembly language.

Example:

C Program

```
int errflag;
extern task();

driver()
{
...
     task (i);

...
}
```

Assembly Language

```
...

XCOM        _errflag,4
SECTION     code,,C
XDEF        _task
...
_task:...
      move.l _errflag,d0
...
```

The C compiler reserves some identifiers beginning with two underscores for internal use. Avoid using identifiers with two leading underscores in assembly routines. Also, avoid using identifiers with one leading underscore in C source files; these identifiers may conflict with internal names.

Some assemblers and linkers translate all lower-case letters to uppercase or vice versa. Since the C language is case-sensitive, check your assembler and linker documentation for information on this topic.

The Microtec C compiler and assembler toolkits retain the original upper- and lower-case status of all letters within a symbol.

Static Variable Names

The compiler automatically prefixes a string to all static variable names declared within a function. The string is **_.S***n*_, where *n* is a number. For example, if the variable name *var* has been declared within a function, it will be changed to **_.S***n_var* in the generated assembly language code.

Line Numbers

If you specify the command line option for debugging, the compiler generates actual C source lines and line numbers and includes them as comments in the assembly language source file.

If the **-GI** option is specified, the compiler generates line number symbols as labels for debugging purposes. These line number symbols have the form **LL***nn*, where *nn* is the line number. Line number information is generated for all executable lines of code.

Code and Data Sections

See Chapter 8, *Run-Time Organization*, in this manual for information on section names, placement, and method of addressing.

Compiler Output Listing

The compiler can generate an output listing file when you use the appropriate command line option (see the section *Generate Listing File* in Chapter 2, *UNIX/DOS User's Guide*, for more information). When a listing is generated, the compiler assigns a line number, beginning with number 1, to each source line in your C module. When files are included in the C source module, the compiler will also assign a number, beginning with number 1, to the include file lines. When the last line of the include file has been numbered, the compiler resumes numbering the source file lines from the line prior to the included file.

The compiler output lists error messages if any occurred during compilation. An arrow points to the error in the listing. The error message is printed immediately below the line where the error was encountered.

The output listing also includes a code summary for each routine and the entire module. These code sizes are estimates since it is not known at compile time whether certain address references will require 16 or 32 bits.

Error Statements: Appendix A

Overview

This appendix lists the messages produced by Microtec compilers. A description of the error and an example of an error are provided whenever possible. Messages have the following format:

"*filename*", line *number* pos *pos_number*; (*severity*) *message*

where *filename* is the name of your file. The location of the error in that file is indicated by the line number (*number*) and column position (*pos_number*). The message severity is shown in parentheses. The severity levels are described in the next section.

Note

This appendix contains explanations of error messages that can occur during the typical operation of this product. However, messages that are outside the scope of this product (e.g., operating system error messages) are not documented in this appendix. Refer to the *Release Notes* for this product for additional information on error messages.

If you encounter an undocumented error message, please contact Microtec Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

The messages are listed alphabetically according to the following rules:

1. The message severity is placed at the end of the message to help you scan alphabetically for the message you want.

2. Messages are listed based on the first significant word of the message (words like “a,” “an,” “the,” etc. are not considered significant) and punctuation is ignored for alphabetizing purposes.

3. If the first word is a variable that is supplied at the time of the message, the message is indexed by the next significant word. For example, the message “*name* `undefined`” would be listed alphabetically under “undefined.”

4. Messages preceded with the word **[ANSI]** indicate that an error is related to an ANSI feature. These messages are listed based on the text of the message.

Your compiler may not generate all of the error messages listed in this appendix.

Message Severity Levels

There are two classes of errors that can occur during linker program execution. The first class is fatal, and processing is abandoned. The second class is nonfatal, and processing proceeds after the error is reported.

Messages are followed with a letter in parentheses indicating the severity type of the message. Table A-1 lists the message severity levels.

Table A-1. Message Severity Levels

Type	Severity	Meaning
(F)	Fatal	Always fatal. Processing aborts. Fatal errors arise from conditions that make further compilation impossible, such as missing source and include files or premature **End-Of-File**. Most often, these errors can be corrected by use of a more complete search path or a quick examination of the files and options used to invoke the compiler.
(E)	Error	Usually fatal, but processing continues for diagnostic purposes. In general, an error message indicates that the compiler has encountered a syntax or logical error. By default, the compiler adopts the view that it should not generate code after an error **(E)** occurs.

(cont.)

Table A-1. Message Severity Levels (cont.)

Type	Severity	Meaning
(W)	Warning	Not fatal. This message indicates code that is syntactically correct, but can result in unexpected program behavior.
(I)	Informational	Informative and often appears in conjunction with another error message. An informational message is similar to a warning message but is less likely to alter user program behavior. By default, informational messages are suppressed but can be displayed by the appropriate compiler option (see the section *Suppressing Error Messages* in this chapter).

Error Position Marker

For a line of C code containing an error, the compiler puts a circumflex (^) either directly under the offending element or as close to the problem as possible.

Example:

```
1     main()
2     {
3           int i
4           printf("hello, world\n";
            ^
>> (E) syntax error; unexpected symbol: "printf"
5     }
```

This example shows how the omission of a semicolon on line 3 generates an error message with the error position marker pointing to the first statement after the error occurred.

Suppressing Error Messages

You can suppress error messages of a certain severity level and below using the compiler options shown in Table A-2. Fatal errors cannot be suppressed.

Table A-2. Using Compiler Options to Suppress Diagnostic Messages

Option	Suppresses
-Qi (default)	(I)
-Qw	(I), (W)
-Qe	(I), (W), (E)
-Qs	summary

By default, only informational messages are suppressed. There is a limit to the number of **(E)** messages that can be produced in every compilation. This limit is 20 by default but can be changed with the **-Q***number* compiler option. By setting *number* to zero (0), compilation stops if any **(E)** errors are detected.

Example:

```
1     array[100];
      ^
>> (I) [ANSI] missing type; "int" assumed
2     int *if;
            ^^
>> (E) syntax error; unexpected symbol: "if"
>> (F) too many errors

2 Errors    1 Informational
```

Messages of type **(W)** and **(I)** are not considered in the error count.

Example:

```
1     foo()
2           {
3           undefined[*undefined](undefined);
            ^
>> (E) undeclared identifier "undefined"
4     }
```

```
1 Error
```

Although this example shows several uses of the term `undefined` (all of them errors), the compiler avoids multiple error messages and simply identifies the pri-

mary error rather than print multiple error messages regarding the same error. The final error count is shown in a summary message at the end of the listing.

The next example shows the results when the summary message is not suppressed, warning and informational messages are suppressed, and the error message limit is set to 5. The compiler options are as follows:

```
-nQs -Qw -Q5
```

Example:

```
1     array[100];
2     int *p=7;
3     int vet[10;
                ^
>> (E) syntax error; unexpected symbol: ";"
1 Error  1 Warning (suppressed) 1 Informational (suppressed)
```

The next example shows the results when the summary message is suppressed, warning and informational messages are suppressed, and the error message limit is set to 0. The compiler options are as follows:

```
-Qs -Qw -Q0
```

Example:

```
1     array[100];
2     int *p=7;
3     int vet[10;
                ^^
>> (E) syntax error; unexpected symbol: ";"
>> (F) too many errors
```

The occurrence of any error terminates compilation.

Reporting Problems

Microtec performs extensive tests on its compilers. However, even with the most exhaustively tested software, you may still encounter a problem. When you find a problem, you should first check your *Release Notes* to see if there is any information on your problem. If there is no mention of your problem in the *Release Notes*, please report your problem promptly to Microtec Technical Support. If you can, have a test case prepared.

Preparing a Test Case

A small test case that reproduces your problem can help Microtec Technical Support quickly determine a solution to your problem. Unfortunately, the complicated

combinations of include files and intricate macros that comprise a program can make it difficult to isolate a problem and produce a clean test case.

Therefore, it is important that you provide complete information to Microtec Technical Support.

Calling Technical Support

Follow these steps:

1. Prepare a test case source file that exhibits the problem when compiled.
2. Add comments to your source file to indicate where the problem occurs.
3. Verify that the problem is still there.
4. Call Microtec Technical Support. The Technical Support representative will make arrangements for you to deliver the test case source file.

Compiler Messages

Following are the various possible error messages that the compiler can return, along with an example and short explanation of the error.

does not appear before a parameter; ignored, (W)

The # operator does not appear before a parameter. This message is from the preprocessor.

Example:

```
#define g(b) # y
```

has no left argument; ignored, (W)

The ## operator has no left-hand argument. This message is from the preprocessor.

Example:

```
#define f(a) ## a
```

has no right argument; ignored, (W)

The ## operator has no right-hand argument. This message is from the preprocessor.

Example:

```
#define f(a) a ##
```

& operator has no effect, (W)

The address operator is redundant in the expression. This usually occurs whenever there is an assignment of a function pointer, since the compiler automatically assigns the address of the function.

Example:

```
int foo() { int (*p)(); f(); p = &f; }
```

/* inside a comment, (I)

Comments cannot be nested. This informational message indicates that you are trying to nest comments; this occurs when the **-Qi** compiler command line option to allow informational messages is set. This message is from the preprocessor.

Example:

```
/* /* inner */ */
```

address arithmetic performed with "int" precision, (W)

A **long** value is used as a subscript or added to a pointer. The value is truncated to 16 bits of precision. This warning applies to processors where **sizeof(int) < sizeof sizeof(long)**.

Example:

```
int a[100];
long i;
a[i]=0;
```

address expected in initialization of *variable*, (W)

Whenever a pointer is initialized by a nonpointer value that is a character or an integer constant, the compiler generates the initializing assignment and issues this warning.

Example:

```
int *p = 0XFFFF;
```

address/pointer extended, (I)

The size of a pointer or an address has been extended. This message is an informational message produced only if the extra checking option is specified on the command line (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
#pragma option -v -nQ
(long) &c; /* informational message issued. */
```

This statement attempts to extend the size of the address of variable `c` to the size of `long`.

adjacent string literal concatenation conflicts with "no ansi" option, (W)

The concatenation of adjacent strings is an ANSI feature. Your file has been compiled with ANSI features disabled (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA
char *str = "abc" "cdf";
```

aggregate initialization conflicts with "no ansi" option, (W)

Aggregate initializations are ANSI extensions. Your file has been compiled with the

ANSI features disabled, but the compiler found a function with an aggregate initialization (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2,

UNIX/DOS User's Guide, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA
struct S { int i; int j; } s0 = {1, 2};
struct S s1 = s0;
```

always false, (W)

If an expression will always evaluate to false, this warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
#pragma option -v
main(i)
{
     if (sizeof(int)==sizeof(char));     /* always false */
}
```

always true, (W)

If an expression will always evaluate to true, this warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
#pragma option -v
main(i)
{
     if (sizeof(int)==sizeof(int *));    /* always true */
}
```

[ANSI] argument is incompatible with formal parameter type, (W)

Whenever the arguments passed to a function do not match the function prototype specification (e.g., passing pointers when integers are expected and vice versa), a warning is issued to highlight the type incompatibility.

Example:

```
int foo(int i);
int baz(int *p);

int bar() { int *v; foo(v); baz(100);}
```

argument is void, (E)

The compiler attempts to apply the C type conversion rules whenever it finds function arguments that do not match the function prototype. In K & R C, no function prototypes are available, so the compiler applies default conversions to type **int**. However, when a function returning a **void** is used as an argument, the compiler cannot make any conversions; hence, the compiler generates an error that the argument is of type **void**.

Example:

```
void foo();
void zap();

int bar() { int i; zap(foo()); }
```

arithmetic type expected, (E)

The compiler checks to see if the expression that is evaluated contains types on which arithmetic operations can be performed, such as **char**, **int**, **long**, **double**, and **float**. Only addition and subtraction operations are defined for pointers. Other operations on pointers, like unary operations, multiplication, division, etc., are incorrect.

Example:

```
int foo() { int *r = 0; int k; k = (int)(r*4); }
```

array has invalid null dimension, (E)

A function parameter or local variable was declared as an array with the null array dimension.

Example:

```
void f(int ia[][])     /* error */
{
     int arr[][];     /* error */
}
```

assembler inlining function incompatible with binary object output, (E)

This message indicates that the **asm** pseudofunction cannot be used with binary object output. The usage of **asm** implies an assembly step.

Example:

```
f() { asm(); }
```

assembler inlining function interpreted as user function due to "no MRI extension" option, (I)

ASM directives are only supported if the Microtec extensions option flag is on. This is a default option. Negate this option to compile the file with no Microtec extensions (see the **-x** option in section *Enable Microtec Extensions* in Chapter 2, *UNIX/DOS User's Guide*, for further information about disabling Microtec extensions).

Example:

```
#pragma option -nx
int foo() { ASM("4"); }
```

assignment to array or function, (E)

In most cases, the compiler treats an array as if it were equivalent to a pointer; however, it does not permit an assignment to an array.

Example:

```
int foo() { int a[10]; int *p; p = a; a = p; }
```

[ANSI] attempt to modify a "const," (E)

A variable declared with the **const** qualifier was used as an lvalue, thus there was an attempt to modify the **const** variable.

Example:

```
int foo() { const int x; x=5; }
```

auto allocation for *XXXX* exceeds 32K, (E)

The total size of local variables for function *XXXX* is larger than 32K. This limitation is imposed by the addressing of the frame pointer with a negative offset.

To resolve this problem, reduce the size of local variables by declaring the non-recursive variables using the **static** keyword.

break statement is not inside a loop or switch, (E)

A **break** statement has appeared outside of a **while**, **for**, **do**, or **switch** statement.

Example:

```
int foo() { break; }
```

call of a nonfunction, (E)

A function name is treated as a pointer to the function of its return type. The compiler allows calls to be made only if it is a pointer to a function.

Example:

```
int foo() {void *vvar = 0; void (*vfunc)(); vfunc(); vvar();}
```

can't open file *filename*, (F)

The compiler cannot open the named file, either because the input file does not exist in the specified directory or, if the file exists, because the compiler encountered an access error. On UNIX systems, the error is due to an incorrect read permission on

the file.

can't open *filename* for read, (E)

filename does not exist.

can't open *filename* for write, (E)

The code generator cannot open the *filename* for output (source or mixed assembled listing or object file). The most common reason is that you are out of directory entries or disk space (DOS) or do not have write permission (UNIX).

can't open #include file *file*; directive ignored, (W)

The specified include file *file* cannot be opened. This message is from the preprocessor.

Example:

```
#include "XYZ:FILE"
```

can't open output file *file*, (F)

If you invoke the preprocessor on the command line and direct the output to a file that the preprocessor cannot write to, you will get this fatal error message.

Example:

If you invoke the preprocessor, specify `/ouch.i` as the output file, and you do not have write permission to the / directory, you will get the following error message:

```
(F) can't open output file /ouch.i
```

can't open source file *file*, (F)

This message is from the preprocessor.

Example:

If you invoke the preprocessor, specify **ab.c** as the input file, and **ab.c** does not exist, you will get the following error message:

```
(F) can't open source file ab.c
```

can't write to output file: "*filename*", (F)

The compiler cannot write an intermediate, temporary file. Check to see if you have the following conditions:

- File names such as *file*.**1** or *file*.**2** where *file* is the name of the input source file

- Enough space in the temporary directory for the temporary files

cannot exec *code_generator*: error *nn*, (E)

When the global optimizer runs out of memory, the code generator tries to invoke itself to compile the program again without global optimization. This error occurs when the code generator is unable to invoke itself. Refer to **error.h** for an explanation of the error code *nn*.

cannot negate option *option_name*, (W)

A command line option that does not have a negative form was specified on the command line with a negative form.

Example:

```
mcc68k -nFlp test.c
```

cannot reserve more than *number* register(s) of class *class*, (W)

The compiler will not allow you to reserve more than *number* registers with command line options. See the section *Produce Minor Code Generation Variations* in Chapter 2, *UNIX/DOS User's Guide*, for more information.

cannot reserve *reg_name*, (W)

The compiler cannot reserve register *reg_name* because it is required to adhere to function calling conventions.

cannot shorten relocatable to less than *x* bytes, (W)

No pointer can be shortened to less than the size of the smallest available pointer. *x* represents the minimum number of bytes.

Example:

```
int near a;
```

```
main (void) { (char) &a; }
```


case label is not inside a switch statement, (E)

case is a reserved keyword and can only appear within an enclosing **switch** statement.

Example:

```
int foo() { int i; case 0: i = 1; }
```

cast of a nonscalar type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can only occur on scalar types like integers, pointers, etc. A variable of **struct** type is not a scalar type and hence cannot be an rvalue (right hand of assignment) in a cast expression.

Example:

```
struct tag { int i; } s;
int foo() { int i; i = (int)s; }
```

cast of void type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can only occur on scalar types like integers, pointers, etc. A **void** type cannot be used in a cast expression.

Example:

```
int foo() { double x; x = (double)(void)0; }
```

cast to a nonscalar type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can only occur on scalar types like integers, pointers, etc. A variable of **struct** type is not a scalar type and hence cannot be an lvalue (left hand of

assignment) in a cast expression.

Example:

```
struct tag { int i; } s; int foo() { int i; s = (struct tag)i;
}
```


[ANSI] cast used as an lvalue, (E)

Casts that would potentially change the underlying type of lvalues (the left hand of an assignment) may lead to undefined behavior. Therefore, the compiler generates a warning if your code is compiled with the strict ANSI option (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -A -nx
int foo() { unsigned short ch; (signed int)ch = -1; }
```

Compilation aborted, (E)

You have hit **CTRL-C** key is hit during compilation.

compilation terminated: *reason*, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Technical Support with your test case and the error message that was generated.

compiler and input hopelessly out of sync, (F)

An unrecoverable syntax error was encountered. The error was severe enough to prevent the compiler from parsing the rest of the program. Correct the error and recompile.

constant *constant* exceeds machine capacity; truncated, (W)

Each machine places a limitation on the integer value that can be represented. If the integer specified exceeds the machine capacity, the compiler generates a warning.

Example:

```
int i = 0x100000000;
```

constant expression expected, (E)

The expression associated with a **case** label in a **switch** statement needs to be a constant expression that can be evaluated at compile time. The compiler does constant folding and evaluates the expression during compile time.

Example:

```
switch(i-j);
```

constant in string *string* exceeds machine capacity; truncated, (W)

Each machine places a limitation on the integer value that can be represented. If the integer within a string exceeds the machine capacity, the compiler generates a warning.

Example:

```
char *p = "str\x100000000";
```

continue statement is not inside a loop, (E)

A **continue** statement has appeared outside of a **while**, **for**, or **do** statement.

Example:

```
int foo() { continue; }
```

data allocation for *XXX* exceeds 64K, (E)

The size of an aggregate is larger than 64K.

data allocation for *XXX* segment exceeds 64K, (E)

The total size of data allocated for the particular segment is larger than 64K. Force the large aggregates to far segments by explicitly using the **far** keyword or specifying the big data option (**-Zb**). *XXX* is one of the following: **bss**, **const**, or **data**.

Example

```
char aa[40000];
char bb[40000];
```


default label is not inside a switch statement, (E)

default is a reserved keyword and can only appear within an enclosing **switch**.

Example:

```
int foo() { default: ; }
```

#define or #undef of *identifier* ignored, (W)

A **#define** or **#undef** statement is ignored. The preprocessor ignores the following **#undef**s:

__DATE__
defined
__FILE__
__LINE__
__STDC__
__TIME__

Example:

The preprocessor ignores the following `#define` macro `defined`:

```
#define defined
```

division by 0; result assumed 0, (W)

A division by zero was attempted. This message is from the preprocessor.

Example:

```
#if 1/0
```

division by zero, (W)

Division by zero in many computer architectures results in an arithmetic exception. However, if undetected, division by zero errors may lead to undefined execution

However, if undetected, division by zero errors may lead to undefined execution behavior. The compiler tries to detect if the divisor in a divide operation is zero; if so, it generates a warning.

Example:

```
int foo() { int i; i = 4/0; }
```


duplicate case label *number*, (E)

The constant expression associated with a **case** label should evaluate to unique constant integer values. Any duplication of **case** label expressions is reported by the compiler.

Example:

```
int foo()
{
      int i;
      i = bar();
      switch (i) { case 1: break; case 3-2: break; }
}
```

duplicate default label, (E)

There can only be one default label associated with a **switch** statement. Any duplication of default labels is reported by the compiler.

Example:

```
int foo()
{
      int i;
      i = bar();
      switch (i) { default: break; default: break; }
}
```

duplicate statement label *label*, (E)

Statement labels should be unique within a function scope. Any duplication of a statement label is reported by the compiler.

Example:

```
int foo()
{
      int i;
      i = bar();
      a : if (i == 0)    {
```

```
                                        a: ;
                                        }
}
        int bar()
{
        int r; a:;
}
```


#elif used outside #if/#endif pair; ignored, (W)

An **#elif** statement does not occur within a **#if/#endif** sequence. This message is from the preprocessor.

Example:

```
#elif 1
```

#else used outside #if/#endif pair; ignored, (W)

An **#else** statement does not occur within a **#if/#endif** sequence. This message is from the preprocessor.

Example:

```
#ifdef __STDC__
     volatile int a;
#endif
#else
```

empty character literal, (W)

Empty character literal symbols were used. This message is from the preprocessor.

Example:

```
int a = '';
```

[E] empty conditional statement

This warning message is reported when the **-v** option for extra checks is enabled and there is an **if** statement with a null statement in the conditional part.

Example:

```
main(){
    if (1) ;
}
```

[ANSI] empty declaration, (W)

A declaration is missing an identifier. Fix the error by declaring an identifier.

Example:

```
int;
```

empty hexadecimal constant; "0x0" assumed, (W)

The hexadecimal prefix appears without an associated number. This message is from the preprocessor.

Example:

```
int a = 0x;
```

[ANSI] empty hexadecimal sequence; 'x' assumed, (W)

The escape sequence **\x**, etc. is reserved in ANSI; therefore, the escape sequence is ignored. This ANSI behavior can be suppressed if you compile your program with the ANSI features disabled (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for more information about enabling and disabling ANSI features).

Example:

```
int foo() { char p; p = '\x'; }
```

#endif does not have a matching #if; ignored, (W)

A **#endif** statement does not have a matching **#if** statement. This message is from the preprocessor.

Example:

```
#if __STDC__
     volatile int a;
#endif
#endif
```

enum value is not a valid constant, (E)

Only constant expressions can be used to assign enumerator values. The compiler performs constant folding and evaluates the expression during compile time.

Example:

```
int foo() { int x = 1; enum { a =0, b = x }; }
```

[ANSI] enumerated data type in bit-field declaration; accepted, (W)

An enumerator was used in a bit field specification. Bit fields are specified for integers only.

Example:

```
#pragma option -nx
struct S { enum color e1 : 3; };
```

#error: *message*, (F)

The **#error** directive is used. This directive causes the ANSI C-compliant preprocessor to issue a diagnostic message that includes the tokens specified in the **#error** directive. This message is from the preprocessor.

Example:

```
#error oops
```

[ANSI] escape sequence '*character*' reserved for future use; accepted as '*character*,' (I)

The escape sequence *character*, etc. is reserved in ANSI; therefore, the escape sequence is ignored. This ANSI behavior can be suppressed if you compile your program with ANSI features disabled (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for more information about enabling and disabling ANSI features).

Example:

```
int foo() { char *p; p = "\s"; }
```


extra files specified; remainder ignored, (W)

The preprocessor was invoked with an extra (third) file specified on the command line.

[ANSI] extra parameter(s) following a void prototype, (E)

This is a function prototype specification error. Only function pointers of type **void** can be passed as function arguments. Fix the error by including a parameter name for the **void**.

Example:

```
int foo(void , int );
```

far pointer/address truncated to near pointer/address, (I)

When you convert a far pointer to a type that is only large enough to hold a near pointer, the compiler will convert the far pointer to **near**. The **-nQ** option must be activated to trigger this message.

Example:

```
int far F();
int a = (int) F;
(near *a2)() = F;
```

fewer arguments than formats, (W)

A format string has fewer arguments than the number of parameters passed to the I/O statement. The message is only available under the **-v** option for extra checks.

Example:

```
printf("%d %d %d %d", 1, 2);
```

file not found *filename*

The compiler cannot find the specified file.

The compiler cannot find the specified file.

floating-point overflow/underflow, (E)

This message is issued if a floating-point value is outside of the floating-point range.

format/argument mismatch, (W)

A format string does not match the type of the corresponding I/O argument. This message will be a warning **(W)** if the I/O statement is likely to produce the wrong result; the message will be informational **(I)** if the I/O statement would probably produce the correct result but would print a wrong result on another target. This message is only available under the **-v** option for extra checks, and the informational messages are suppressed unless you enable them with the **-nQ** option.

Example:

```
printf("%f, 1");
printf("%d, 1L");
```

function *identifier* declared as *keyword*, (E)

The **auto** and **register** keywords apply to variables and specify a location. They cannot be used to specify the return type in a function declaration nor in its parameter types.

Examples:

```
int foo() { auto int g(); }
```

function *function* has incomplete return type, (E)

A function can only return with a complete return type although the return statement is not explicitly stated. The compiler tries to return the correct type but finds the type has not been fully specified; hence, the error.

Example:

```
struct S foo() { }
```

function *function_name* is redefined, (E)

A function can be defined only once although it can be declared a number of times. In C, the file is the separate compilation module; therefore, only the linker can report if a function has been defined in more than one file.

Example:

```
int bar();
int bar();
int foo() { int i; i = 0; }
int foo() { int i; i = 0; }
```


GOTO into a LOOP, (W)

This message is produced if a **GOTO** statement causes program execution to branch into the body of a loop. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
goto_into_loop()
{
int i;

      if (x)
            goto L3;
         for (i=0; i<=4; i++)
      {
            xxx();
L3:
            yyy(i);
      }

}
```

GOTO into a SWITCH statement, (W)

This message is produced if a **GOTO** statement causes program execution to branch into the body of a **SWITCH** statement. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
goto_into_switch()
{
      if (x)
            goto L3;
      switch(x=test())
      {
      case 1:test(x); break;
      case 2: xxx(x); break;
L3:
```

```
        case 3: yyy(x); break;
        case 4: zzz(x); break;
        }
}
```


identifier *identifier* declared as void, (E)

A formal parameter was incorrectly declared to be of type **void**.

Example:

```
int foo(int x, void y);
```

identifier expected; assumed undefined, (W)

#ifdef should be supplied with an identifier as a parameter. If it is incorrectly supplied, the **#ifdef** construct is assumed to be undefined; hence, the enclosing code between the **#ifdef** and its corresponding **#endif** is left out. This message is from the preprocessor.

Example:

```
#ifdef 1
int i;
#endif
```

#if does not have a matching #endif; supplied, (W)

A **#if** or a **#ifdef** should have a corresponding **#endif** construct. This message is from the preprocessor.

Example:

```
#if 1
int i;
```

IF expression evaluated to constant *number*, (W)

If an **IF** expression will always evaluate to a constant, this warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
#define FLAG 0

if_exp_evaluated_to_const()
{
int i = FLAG;

      if (i)       /* i == 0 */
            test();
}
```

In this example, the **IF** expression evaluates to a constant 0.

illegal argument to predefined macro *macroname*, (W)

The predefined preprocessor macro does not accept the argument supplied. This message is from the preprocessor. Refer to the section *Preprocessor Macros* in Chapter 3, *The Preprocessor*, of this manual for more information.

Example:

```
__OPTION_AVAIL (a b c d e f g);
```

illegal char in macro definition; symbol *symbol* deleted, (W)

A comma was expected; the unexpected symbol was deleted to correct the situation. In the following example, **f(a)** and **f(a;)** both yield **abc**. This message is from the preprocessor.

Example:

```
#define f(a;) abc
```

ill-formed number, (E)

A preprocessing number is not a valid number.

Example:

```
int x = 1A;
```


incompatible type returned, (E) or (W)

The type returned is incompatible with the declared type of the function.

Example:

```
int foo() { char *p = "abc"; return p; }
struct tag { int i; }  bar() { return 0; }
```

inconsistent packing attributes for tag "*tag*"; "packed" prevails, (W)

A structure has been declared as both packed and unpacked.

Example:

```
struct tag;
packed struct tag;
```

inconsistent packing attributes for tag "*tag*"; size would change from *size1* to *size2*, (E)

A structure has been declared inconsistently. The conflicting sizes are shown in bytes.

Example:

```
struct tag {char a; short sh;};
packed struct tag {char a; short sh;};
```

inconsistent redeclaration of *identifier*, (E)

An identifier of a declared type cannot be redeclared to be of a different type. The compiler generates an error message whenever it detects this condition.

Example:

```
int x; char x;
```

This error will also occur if a function prototype uses a structure that has not been previously declared. The scope of the structure terminates at the end of the proto-

type definition, and a later definition of the structure is interpreted as a redefinition

of that structure. To avoid this problem, declare the structure prior to the function prototype.

Example:

```
extern void f(struct s);
...
struct s{         /* interpreted as redefinition of s */
      ...
}
```

inconsistent redeclaration of tag *tag*, (E)

A structure tag of a declared type cannot be redeclared to be a structure tag for a different structure. The compiler generates an error message whenever it detects this condition.

Example:

```
struct S { int x; }; struct S { char ch; };
```

inconsistent use of tag *tag*, (E)

A tag of a declared type cannot be used as a tag for a different type. The compiler generates an error message whenever it detects this condition.

Example:

```
struct S ; enum S;
```

increment/decrement of array or function, (E)

Pointer arithmetic operations cannot be performed on an array or function pointer.

Example:

```
int foo() { int a[10]; a++; }
```

inefficient alignment (offset for member "*variable*" not a multiple of *number*),

inefficient alignment (offset for member *variable* not a multiple of *number*), (W)

A misaligned member of a packed struct will be loaded and stored inefficiently. The access will generally be "burst" into a byte-by-byte or word-by-word load or store. On the 68000 and 5200, the bursting is done by the compiler, but on the 68020 and

other machines that support misaligned access, the chip handles the bursting transparently. In either case, the burst access is less efficient than an aligned access.

Note that this warning is issued only when parsing a declaration of a packed struct and is not issued in all cases of misaligned access. For instance, the 68000 compiler bursts access to all fields on an element of an array of packed structs (since the alignment of any element is generally unknown), but the warning is not issued.

[ANSI] inefficient evaluation in "double" because of "double" constant, (I)

Whenever double constants are used instead of integer constants, the compiler generates an informational message to indicate that the conversion is an inefficient one.

Example:

```
int foo() { float fl = 0; fl = fl + 1.0; }
```

int expected in initialization of *variable*, (W)

The initializer supplied in an initializer list should be type compatible with the variable it is expected to initialize.

Example:

```
int i = &i;
```

int overflow for "*number*," (W)

This message is issued if an integral value outside of the integer range is supplied.

Example:

```
void f(void) {
     0100000000000;
     4573741824;
     0xFF0000000;
}
```

int overflow in string "*string*", (W)

This message is issued if you supply an integral value in a quoted string, and the integral value is outside of the integer range.

Example:

```
void f(void) {
     "string\xaaaaaaaaaaaaa";
}
```

int type expected, (E)

The compiler expects an **int** type or a type that can be implicitly converted to an integer.

Example:

```
int foo() { int i = ~1.0; }
```

int value expected; real value truncated, (W)

Whenever an lvalue is of type **int** and the rvalue is of type **float** or of an arithmetic type that is larger than the **sizeof(int)**, the compiler generates a warning that the rvalue is being truncated to suit the **sizeof** the integer. This truncation can lead to unexpected run-time behavior.

Examples:

```
int foo() { int array[3.4]; }
int foo() { int *p; float q; p+q; }
```

integer line number expected; directive ignored, (W)

A line directive must contain a reference to a line number. This message is from the preprocessor.

Example:

```
#line =
int i;
```

Internal Compiler Error: *reason*, **(F)**

This error is caused by an internal problem with the compiler. Please contact Microtec Technical Support with your test case.

internal error: *reason*, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Technical Support with your test case.

interrupt function has parameter(s), (W)

An interrupt function cannot have parameters. The compiler will generate a warning whenever it detects such a condition.

Example:

```
void interrupt foo(int i);
```

interrupt function returns non-void type, (W)

An interrupt function must always return a **void**. The compiler will generate a warning whenever it detects a return type other than **void**.

Example:

```
int interrupt foo();
```

interrupt procs not supported, (W)

The compiler does not support declaring functions as interrupt handlers with the **interrupt** keyword in this release.

Example:

```
int interrupt foo();
```

invalid argument for assembler inlining function, (E)

An invalid argument was used with the **asm** pseudofunction.

Example:

```
f() { asm(3) }
```

```
f() { asm(3); }
```


invalid array component type, (E)

An array cannot be of type **void**.

Example:

```
void arry[1000];
```

invalid array size, (E)

The size of an array should be a positive integer and the result of a constant expression. The compiler evaluates the expression and checks to see that it is a positive integer. Any incorrect size specification will result in this error.

Example:

```
int arry[-1000];
```

invalid digit in octal constant *constant*; assumed decimal, (W)

In C, all octal numbers are identified by a leading zero and must contain only octal digits. This message is from the preprocessor.

Example:

```
int a = 08;
```

invalid #include file name *symbol*; directive ignored, (W)

An unacceptable file name has been chosen for an include file. This message is from the preprocessor.

Example:

```
#include 123
int i;
```

invalid initialization for *identifier*, (E)

The initialized variable requires an initializer of the same type. The compiler attempts to convert the types if a type conversion is possible. Otherwise, the com-

piler reports an error that the initialized variable has been supplied with an invalid initializer.

Example:

```
void (*p)() = 0;
int a[10] = p;
```

invalid insert *symbol* for assembler inlining function, (E)

An **ASM** instruction cannot contain references to undeclared variables in the C program. An **ASM** instruction should also be syntactically correct. If the variable is undeclared or a syntax error occurs within the **ASM** string, the compiler generates an error message.

Example:

```
int foo() { ASM("mov `undef`, `truncat"); };
```

invalid return type, (E)

The function return type cannot be an array. The compiler will generate an error whenever it detects this condition.

Example:

```
typedef int A[10];
A foo();
```

invalid size for bit-field *bit-field_name*, (E)

Bit field members cannot specify a field size greater than the size of the type to which the bit field belongs.

Example:

```
struct S { unsigned int a : 100; unsigned int b; };

/*
```

```
struct S1 { unsigned int a : sizeof(unsigned int) + 1;
unsigned int b; };
*/
```


[ANSI] invalid specifier(s) used with "void", (E)

No additional type specifiers are used along with the type **void** except for **const** and **volatile**. The compiler generates an error (ANSI or otherwise) if it detects such a condition.

Example:

```
unsigned void* a;
```

invalid storage class *storage_class* for parameter *parameter*, (E)

Function parameters are formals and cannot be associated with any storage specifications.

Example:

```
int f(static int i);
```

invalid storage class *storage_class* for parameter *parameter*, (W)

An invalid automatic storage class is being specified for a formal parameter.

Example:

```
int foo(x) auto x; { }
```

invalid suboption "*sub_option_name*" given for *option_name*, (W)

The compiler does not recognize the suboption entered on the command line.

Example:

```
-Fz
```

invalid type for member *member_name*, (E)

The members of a structure should be a valid type. Valid types include the built-in

C types and any user-defined **struct**s, **union**s, or **enum**s. Members cannot be functions or of type **void**.

Example:

```
struct S { void b; int i; };
```


invalid value "*value***" given for** *option_name***, (W)**

An invalid value was given for a compiler command line option.

Example:

If you specify **-Za3** on the command line, you will receive the message:

```
(W) invalid value "3" given for -Za
```

identifier **is not a parameter, (E)**

The parameter names in a function definition should match those in their declaration. The compiler detects an error if the match fails.

Example:

```
int foo(a) int b; { int k; }
```

option **is not possible if there are functions whose total parameter size is >** *number***, (W)**

Your file has been compiled with the compiler command line option to reserve a register, but this register is needed in order to adhere to function calling conventions.

option **is not possible if there are functions with** *n* **parameters in the file to be compiled, (W)**

Your file has been compiled with the compiler command line option to reserve a register, but this register is needed in order to adhere to function calling conventions.

[ANSI] keyword "*keyword***" in empty declaration, (W)**

You cannot specify a storage class or qualifier for a **struct**, **union**, or **enum** tag declaration without declaring an identifier. Fix the error by declaring an identifier for

that structure. The keyword **volatile** in an empty declaration causes a fatal error since the structure declared is not considered volatile.

Example:

```
static struct tag {int i; };
```


lexical error -- unexpected *cause*, (E)

An error was detected during the lexical analysis compilation phase. A short description of the lexical error (*cause*) is provided with the message.

local "extern" variable *identifier* is initialized, (E)

External variables declared in a function scope cannot be initialized. The compiler generates an error message when it detects this condition.

Example:

```
int foo() { extern int e = 0; }
```

[ANSI] local function *identifier* declared as "static," (I)

Whenever a function is declared **static** in a function scope, an informational message is generated.

Example:

```
int foo() { static f(); }
```

local variable *variable* never used, (W)

This message is produced when a local variable is never used or can be replaced by a constant. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
var_never_used()
{
int i, j;         /* all references of i are eliminated */

     i = 3;       /* value of i is not used; assignment
                  /* eliminated */
     x = i + 2;   /* i replaced by 3 */
```

```
}
```

"long double" interpreted as "double," (W)

A declaration with **long** and **double** type specifiers does not increase the precision of the type. Therefore, the compiler generates a warning that these type specifiers

are interpreted to be of type **double**. This message is issued if the extra checking compiler command line option is turned on (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
#pragma option -v
     long double f;
```

In this example, the data type for `f` is declared as `long double`. The compiler treats its data type as **double**.

"long float" interpreted as "double," (W)

A declaration with **long** and **float** type specifiers does not increase the precision of the type. Therefore, the compiler generates a warning that these type specifiers are interpreted to be of type **double**.

Example:

```
#pragma option -v
long float f;
```

"long" truncated to "int," (W)

This message is issued when the compiler converts a variable of type **long** to **int**.

Example:

```
void g(void)
{
     int *p;
     long l;
     unsigned long ul;

     p+l;        /* warning */
     p+ul;       /* warning */
     p[l] + 1;   /* warning */
}
```

macro *name* invoked with wrong number of arguments, (E)

A macro has been invoked with an incorrect number of arguments. This message is from the preprocessor.

Example:

```
define f(a) a+1
void main() {
      int k = f(1,2);
      int j = f(1);
}
```

macro name *symbol* is not an identifier; directive ignored, (W)

The name given in a macro is not an identifier. This message is from the preprocessor.

Example:

```
#define 0
#define $6.0-U.S-Dollars
int j;
```

macro parameter *symbol* is not an identifier; ignored, (W)

The parameters of a macro should be identifiers. This message is issued if any of the macro parameters are not identifiers. This message is from the preprocessor.

Example:

```
#define f(0)
void main() {
      f(0);
}
```

meaningless "far" qualifier; ignored, (W)

Although C grammar allows a specifier list to be attached to any identifier, structure

member name, function name, etc., some specifier combinations will not appear meaningful in the given context. The keyword **far** should be applied only to pointers or objects.

meaningless "interrupt" qualifier for *identifier*; ignored, (W)

Although C grammar allows a specifier list to be attached to any identifier, structure member name, function name, etc., some specifier combinations will not appear meaningful in the given context. The keyword **interrupt** should be applied only to functions.

Example:

```
int interrupt a;
```

meaningless "near" qualifier; ignored, (W)

Although C grammar allows a specifier list to be attached to any identifier, structure member name, function name, etc., some specifier combinations will not appear meaningful in the given context. The keyword **near** should be applied only to pointers or objects.

[ANSI] missing definition for static function *identifier*, (E) or (I)

A static function has been declared and used but not defined. This is an error when compiling with the strict ANSI option. If the declared static function is not used and the file has been compiled with the strict ANSI option, an informational message is generated (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -A -nx
static foo(); int bar() { foo(); };
static zap();
```

[ANSI] missing definition for static function *function_name*; "extern" assumed, (W)

If you call a static function whose definition is missing at the end of the compilation,

this warning is issued, and the keyword **extern** is assumed.

Example:

```
static foo(); int bar() { foo(); };
```


missing definition for (unused) static function "*function_name*," (W)

A static function was undefined. This message is issued if the extra checking compiler command line option is turned on (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for more information).

Example:

```
static int f(void);
```

missing exponent in real constant *constant*; E0 assumed, (W)

An exponent has been left off of a real constant. This message is from the preprocessor.

Example:

```
float a = 1.E;
```

missing format specifier, (W)

A format string contains a "**%**" without a following format specifier. This message is available only with the **-v** option for extra checks.

Example:

```
printf("%");
```

missing #include file name; directive ignored, (W)

The include file name is missing in a **#include** directive. This message is from the preprocessor.

Example:

```
#include
void main() {}
```

[ANSI] missing parameter name, (E)

A function definition in ANSI must contain the complete prototype specification. This error is caused by a missing parameter name.

Example:

```
int foo(int ) { }
```


[ANSI] missing prototype for *function*, (W)

A function definition in ANSI must contain the complete prototype specification. This error is caused by a missing prototype.

Example:

```
#pragma option -A -v
int proto2(int a)
{
     printf("no prototype");
}
```

missing return value, (W)

This warning message is produced when a function declared to return a value does not use the return statement consistently. Since a return statement is not required for any function, the compiler does not warn if no return statements are used. However, when a return statement is used in a function, the compiler will issue a warning if all uses of the return statement are not consistent. If any return statement returns a value, a warning is issued unless all return statements return values. If a return statement is used within a function, a warning is issued for every function exit point that does not use a return statement. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
int returnless_func()
{
     if (x)
          return(1);

}               /* missing return statement at end of
                /* function */
```

missing semicolon for last member, (W)

The last member in a structure or union member list must be terminated by a semi-

colon (**;**) in ANSI C. This message is issued if the strict ANSI option is turned on and the Microtec extensions command line option is turned off. For further information, see the **-A** option regarding setting ANSI-compliant modes and the **-x** option regarding setting Microtec extensions in section *Command Line Option Summary* in Chapter 2, *UNIX/DOS User's Guide*.

Example:

```
#pragma option -A -nx
struct {
     char c;
     int u   /* warning */
} sobj1;
```

[ANSI] missing type; "int" assumed, (I)

Whenever a declaration is made, the compiler expects a type specifier. However, if no type specifier is given, the compiler defaults the declaration to that of an integer and generates a warning.

Example:

```
#pragma option -nQ
a;
```

mod by zero, (W)

Division or modulo by zero in many computer architectures results in an arithmetic exception. However, if undetected, it can lead to undefined execution behavior. The compiler tries to detect if the divisor in a divide operation is zero; if so, it generates a warning.

Example:

```
i = i % 0;
```

mod by 0; result assumed 0, (W)

Mod by zero was attempted. This message is from the preprocessor.

Example:

```
#if 1%0
```

module is too large for available memory, (F)

The compiler is usually very efficient about its memory allocation. However, there are certain limiting factors, such as the number of macros defined, etc. The compiler generates this fatal message when it cannot allocate any more memory.

more arguments than formats, (W)

A format string has more arguments than the number of parameters passed to the I/O statement. The message is only available with the **-v** option for extra checks.

Example:

```
printf("%d %d", 1, 2, 3, 4);
```

more than 32767 arguments (compiler limitation), (E)

The compiler limits the number of function arguments to 32767. This is an internal compiler restriction.

more than 8000 labels in one switch (compiler limitation), (E)

The compiler limits the number of **case** labels that can be handled in a **switch** statement. It can process a maximum of 8000 **case** labels within this statement. This error message is generated when the limit has been exceeded.

name *name* specified for filler bit-field; ignored, (W)

The bit field specification lets you specify a bit field from 1 to the **sizeof** the type of its bit field. The bit field 0 is the filler bit field. Filler bit fields are used to pad the bit fields to the correct size. These fields cannot be referenced. Bit field 0 is the leading filler bit field; hence, a reference to it causes the compiler to issue this warning.

Example:

```
struct S { unsigned int a : 0; };
```

"near" and "far" specified through indirection, (E)

A type cannot be qualified with both **far** and **near** keywords.

Example:

```
typedef int far INT;
INT near a;             /* error */
```

near pointer/address extended to far pointer/address, (I)

When you convert a near pointer to a type that is large enough to hold a far pointer, the compiler will convert the near pointer to **far**, using the normal conversion

method for the target processor. The **-nQ** option must be activated to trigger this message.

no member *identifier*, (E)

Whenever a reference is made to a nonexistent member of a structure, the compiler generates an error.

Example:

```
int foo() { struct S { int m; } s0; s0.p = 0; }
```

non-int in bit-field declaration; accepted, (W)

Bit fields are specified for integers only. The compiler accepts some of the bit field specifications but generates a warning.

Example:

```
#pragma option -nx
struct S { char a : 3; };
```

non-interrupt call of interrupt function, (W)

A non-interrupt call to an interrupt function was made.

Example:

```
interrupt void bar();
void foo() { bar(); }
```

non-prototype function declaration has formal parameter(s), (E)

A function declaration that does not have a prototype specification cannot have parameter names in its argument list. An error is generated by the compiler whenever it detects this condition.

Example:

Example:

```
int foo(a,b);
```

obj file write error, disk probably full, (E)

A system error occurred while writing the object file. The disk is probably full.

offset-only address assigned to long pointer, (I)

An “offset only” is being assigned to a full far pointer (segment + offset). A zero segment will be filled in by the compiler.

Example:

```
int far *PTR;
int near *ptr;
PTR = ptr;
```

operand has incomplete type, (E)

A type can be used only after it has been completely specified.

Example:

```
int foo() { struct S s0; s0 = 0;}
```

operand is an address, (E)

An address is incorrectly being used as an operand; this kind of pointer arithmetic is illegal.

Example:

```
int foo() { int x,a; int *p; p=&x+&a; }
```

operand is not an lvalue, (E)

The operand on the left of an assignment is not a location or variable that can be modified.

Example:

```
int foo() { int x; 5=x+2; }
```

operand is pointer to an object of unknown size, (E)

A pointer to an object of unknown size is incorrectly being used as an operand.

Example:

```
int foo() { struct tag *p; p++; }
```


operand is pointer to function, (E)

A pointer to a function is incorrectly being used as an operand; this kind of pointer arithmetic is illegal.

Example:

```
int foo() { int (*p) (); p++; }
```

operand of & is a bit field, (E)

A bit field is incorrectly being used as an operand of the "address of" operator (**&**).

Fields do not have addresses, so the **&** operator cannot be applied to them.
Example:

```
int foo() { struct { int bit:1;} s; bar(&s.bit); }
```

operand of & is a register variable, (E)

A register variable is incorrectly being used as an operand of the "address of" operator (**&**). Register variables do not have addresses, so the **&** operator cannot be applied to them.

Example:

```
int foo() { register x; bar(&x); }
```

operand of & is not an lvalue, (E)

The operand of the "address of" operator is not legal since it is not a value or a location that can be modified.

Example:

```
int foo() { int x; x=&5; }
```

operands of *operator* have incompatible types, (W)

operands of *operator* have incompatible types, (W)

The operands of an expression are incompatible.

Example:

```
int foo() { int x; if (&x==1) x=7; }
```


option -Md requires runtime initialization of pointers, (W)

This warning indicates that a pointer has been assigned a compile-time initializer that will contain an absolute address.

option *option1* is being phased out; use *option2*, (W)

Some options are no longer used by the compiler. Use of these options does not produce the expected behavior. The compiler will warn about the use of the obsolete option.

Example:

```
#pragma option-68332
int i;
```

option *option* is not implemented, (W)

The compiler does not recognize the option entered on the command line.

option *option_name* might provoke inefficient stack alignment, (W)

The specified option limits the size of the parameters passed to the stack.

***option_name* option not available, (W)**

The compiler does not recognize the option entered on the command line.

option *option_name* requires an argument, (W)

A command line option *option_name* that requires an argument was supplied without an argument.

Example:

```
mcc68k -I test.c
```

option *option* **requires an argument; ignored, (W)**

The compiler command line option for generating a listing file has been used without the file name argument. If the file name argument is not specified, the compiler issues this warning message and writes the listing file to standard output (see the **-l**

option in section *Generate Listing File* in Chapter 2, *UNIX/DOS User's Guide*, for more information about generating a listing file).

options *option1* **and** *option2* **conflict;** *option2* **ignored, (W)**

Two options that conflict with each other have been supplied on the command line. The second (rightmost) option is ignored. Refer to section *Command Line Option Summary* in Chapter 2, *UNIX/DOS User's Guide*, for more information about the options.

Out of memory (*address***:***operation***), (E)**

The code generator ran out of memory at location *address* during phase *operation* of code generation. This error should only occur on PC hosts.

To resolve this problem:

- Check to see if the module has extremely large functions (e.g., more than two thousand lines per function, excluding comments).
- Check to make sure the PC has at the minimum amount of memory required (refer to the *DOS Memory Management Supplement* for information). If not, try removing some memory-resident programs.

packed arrays may generate invalid access (member "*name***", (W)**

A field in an array within a packed structure may not be aligned properly (i.e., it is located at an odd address).

Example:

```
packed struct {
      char a[3];
      short s[10];
};
```

packed structs not supported, (W)

The compiler does not support declaring structures as packed with the **packed** keyword in this release.

Example:

```
packed struct S { int i; char c; };
```


[ANSI] parameter declaration outside() in a prototype definition, (E)

A parameter was declared outside of the parentheses in a function prototype declaration.

Example:

```
int foo(int a) int b; { }
```

parameter *identifier* is hidden by redeclaration, (E)

An identifier declared as a parameter has been redeclared as a local variable; this causes a conflict with respect to the retrieval of that identifier in the function scope.

Example:

```
int foo(x) { int x; bar(); }
```

PC-relative branch range exceeded (E)

The compiler generated a branch to an address that is greater than 32K bytes away. This error can occur when a **switch** statement contains several hundred **case** statements. This error can also occur when a call is made to another function within the same module but greater than 32K bytes away when the **-Mcp** option is used.

Calling external functions using PC-relative code causes a similar error to be produced at link time.

pointer or array type expected, (E)

Something other than a pointer or array type variable is incorrectly being used as an operand of unary * or **[ ]**.

Example:

```
int foo() { int x, y; x=*y; }
```

[ANSI] pointer to const becomes unrestricted, (W)

Whenever **const** pointers are passed as arguments to functions that require non-**const** pointers, there is a potential that the called function may change the object

pointed to by the **const** pointer. Therefore, the compiler generates a warning whenever your program is compiled with the strict ANSI option.

Example:

```
#pragma option -A
int bar(int* ip);
int foo() { const int* jp; bar(jp); }
```

[ANSI] pointer to volatile becomes unrestricted, (W)

If a pointer to **volatile** data is passed as an argument to a function that requires a pointer to non-**volatile** data, the **volatile** object pointed to may be optimized if the called function is optimized. Since optimizations should not be performed on data declared as **volatile**, the compiler generates a warning when this condition is detected.

Example:

```
#pragma option -A
int bar(int* ip);
int foo() { volatile int* jp; bar(jp); }
```

potential run-time error: address is not guaranteed to be properly aligned, (W)

The item being accessed does not necessary reside on the appropriate boundary. For example, if padding has not been added to elements in a packed structure to force alignment, elements in that structure might not be aligned on an even boundary.

Example:

```
packed struct PAK {
      int i;
};

packed struct C {
      char c;
      packed struct PAK pak;
```

```
        char cc;
};

void f() {
        packed struct C a[10];
        int i = 8;
        &a[i].pak;              /* WARNING */
}
```


Since `char c` is a single byte that can occur on an even or odd boundary, there is no guarantee that the packed structure `pak` will occur on an even boundary. If you change the definition of `struct C` to avoid the potential alignment problem, the compiler does not emit a warning message. For example, the compiler will not emit the warning if you define `struct C` as follows:

packed struct C {

packed struct PAK pak;

char c;

char cc;

};

potential run-time error: arbitrary expression for pointer requiring multiples of *number*, (W)

A value has been assigned to a variable through pointer dereferencing, and the value is not of the same type as the variable. This message is issued if the alignment requirement for the variable to which the pointer is pointing is greater than one.

Example:

```
char c;
double *dp = &c;        /* warning */
```

In this example, assigning `c` to `dp` may cause a run-time error because the alignment for type `double` (that `dp` points to) is greater than 1. To avoid this message, cast `c` to `double *`.

potential run-time error: improper alignment, (W)

This message is issued when the compiler detects a potential misalignment that could cause a run-time error.

Example:

```
void f(void) {
     short *p=(short *) 7;
}
```

This example is potentially illegal at run time because even-address alignment is required.

potential run-time error: target has more stringent alignment requirements than source, (W)

This message is issued when the compiler detects a potential misalignment that could cause a run-time error.

Example:

```
void f(void) {
     char i; double *dp = &i;
}
```

In this example, assigning `&i` to `dp` may cause a run-time error because the alignment requirement for `double` is greater than the alignment requirement for `char`.

recursive call *recursion* replaced by jump, (I)

This message is produced when a recursive function call is used. The optimizer replaces such function calls with a jump instruction.

Example:

```
tail_recursion()
{
int i = test();

        if (i)
                return(tail_recursion());/* call to itself
*/
     return(1);

}
```

redeclaration of *identifier*, (E)

Redeclaration of identifiers is not allowed. Two identifiers that have the same name cannot share the same scope.

Example:

```
int foo() { int x; int x; }
```


redeclaration of enum constant *constant*, (E)

An identifier appeared in an enumeration list more than once. Enumeration lists provide a convenient way to associate constant values with names, so an identifier name cannot appear twice.

Example:

```
int foo() { enum {x,y,x}; }
```

redeclaration of member *field_name*, (E)

The elements mentioned in a structure (i.e., members) cannot be declared more than once.

Example:

```
int foo() { struct x {int y; int y;}; }
```

redeclaration of parameter *identifier*, (E)

The same parameter cannot be declared more than once in a function definition.

Example:

```
int foo(int x, int x);
```

redefinition of macro *identifier*, (W)

A macro has been defined more than once. All redefinitions (e.g., the second definition and the third definition if you supply three definitions for the same macro name) are ignored. This message is from the preprocessor.

Example:

```
#define x y
#define x z
```

reinitialized variable *variable*, **(E)**

A variable has been incorrectly initialized more than once.

Example:

```
int x=1; x=2; int foo() { }
```


restriction: aggregates cannot be larger than *value*, **(E)**

The compiler imposes a restriction on the size of aggregates that can be allocated. This restriction arises due to architectural limitations in the target machine that are based upon the largest accessible piece of contiguous memory. The compiler does not allow an individual aggregate size to increase beyond this limitation. These limitations are fairly large, and it would be rare to reach this limit, especially with 32-bit architectures.

Example:

```
int foo() {
char a[0x1ffffff] ;
}
```

restriction: bit-field "*field_name***" straddling** *number* **bytes ; previous field padded (W)**

This message is issued when complex bit-field straddling occurs. Simple straddling happens when a bit field of a packed structure straddles a word unit but ends up spanning one word unit. Complex bit-field straddling happens when a bit field of a packed structure is larger than one word and straddles the previous word unit, the word unit with the bulk of the information, and the following word unit.

Example:

```
packed struct S {
     int a : PART1;
     int b : PART2;

};
```

`struct S` has two bit fields: `a` and `b`. Assuming that `PART1` and `PART2` add up to slightly more than one word unit, the two bit fields require at least two words and possibly three (depending on the starting point of `struct S`). For more information on bit fields, refer to *Alignment of Bit Fields* in Chapter 6, *Internal Data Representation*, in this manual.

restriction: identifier longer than *count* **chars, (E)**

restriction: identifier longer than *count* chars, (E)

The maximum identifier length has been exceeded.

restriction: line too long, (E)

The line length of 64K characters has been exceeded.

restriction: string too long, (E)

The length of a string has exceeded the number of bytes allowed by the code generator. Refer to Table 6-1 in Chapter 6, *Internal Data Representation*, in this manual for more information.

restriction: too many arguments (more than *number*), (E)

The compiler imposes a restriction on the number of function arguments.

Example:

```
int foo() {  bar( 0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0
                );
          }
```

restriction: too many case labels (more than 8192) in one switch, (E)

This message is issued if you have more than 8192 cases in your **switch** statement.

restriction: too many nested switch statements (more than 16), (E)

This message is issued if you have more than 16 nested **switch** statements in your code.

restriction: too many symbols, (F)

The compiler imposes a restriction on the number of symbols. The number of symbols permitted is based on the amount of string space symbols occupy. The maximum amount of string space the compiler can handle is approximately 255 * 64K bytes.

restriction: too many symbols for intermediate file, (F)

Compilers that do not support multiple string tables have a limit of 64K symbols.

scalar type expected, (E)

A variable of nonscalar type was used where one of scalar type was expected.

Example:

```
int foo() { struct {char x;} s; bar(s+1); }
```

segment-offset address assigned to short pointer, (W)

A full far pointer (segment + offset) is being assigned to an "offset only" pointer. Thus, the segment information is lost.

Example:

```
int far *P;
int near *p;
p=P;
```

shortening address may lose significance, (I)

The size of a pointer or an address has been shortened. This message is an informational message produced only if the extra checking option is specified on the command line (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
#pragma option -v -nQ

void f(void) {
      double x;
      char c;
      (long) &c;/* INFORMATIONAL: address/pointer extended */
      (char) &x;/* INFORMATIONAL: shortening address of &x */
      (char) &c;/* INFORMATIONAL: shortening address of &c */
}
```

In this example, the size of the address of `c` has been extended to the size of `long`, which causes the compiler to generate the message "address/pointer extended." The

next two statements shorten the sizes of the addresses of `x` and `c` to the size of `char`, which causes the compiler to issue the message **shortening address may lose significance**.

[ANSI] "signed" and "unsigned" conflict, (E)

Two contrasting declarations (**signed** and **unsigned**) apply to this identifier. To correct the error, only specify one **signed** or **unsigned** for the declaration.

Example:

```
int foo() { signed unsigned int x; }
```

size of a bit-field is undefined, (E)

The **sizeof** operator cannot be applied to a bit field.

Example:

```
int foo() { struct {int bit:1;} s; int x; x=sizeof(s.bit); }
```

size of a function is undefined, (E)

The **sizeof** operator cannot be applied to a function.

Example:

```
int foo() { typedef bar(); int x; x=sizeof(bar); }
```

size of void is undefined, (E)

The **sizeof** operator cannot be applied to **void** since **sizeof(void)** is undefined.

Example:

```
int bar() { int x; x=sizeof(void); }
void *vp;
struct tag;
foo()
        {
        sizeof(void);
        sizeof(struct tag);
```

```
sizeof(*vp);
vp + 2;
vp - vp;
}
```


sizeof applied to an incomplete type, (E)

The **sizeof** operator was applied to an identifier of an incomplete type. This unary operator is used to compute the size of any object that has a known size; in this case, the size is unknown.

Example:

```
int foo() { struct tag s; int x; x=sizeof(s); }
```

source file contains no declarations, (W)

A source file is expected to contain at least one declaration. The compiler generates a warning if it finds a source file with no declarations.

Example:

```
int main()
{
   ;
}
```

specifier specified twice, (W)

A type specifier is incorrectly being used twice in a declaration.

Example:

```
int foo() { const int const x; }
```

specifier specified twice through indirection; ignored, (W)

An identifier has been used twice with the same type qualifier (**const** or **volatile**) in separate declarations.

Example:

```
int foo() { typedef const int INT; const INT x; }
```

specifiers *specifier1* and *specifier2* conflict, (E)

Two or more specifiers with conflicting or undefined meanings were given; the compiler does not discard any of the options and gives an error.

Example:

```
int foo() { short char x; }
```

specifiers "short" and "long" conflict, (E)

Two contrasting declarations (**short** and **long**) apply to this identifier; the **short** declaration is always discarded.

Example:

```
int foo() { short long int x; }
```

statement has no effect, (W)

This warning message indicates that a statement has been declared that will have no effect once executed. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
no_side_eff_statement()
{

int *p;
int x,y,z;

      *p;   /* no effect */
      p++;  /* no effect */
      if (x == y)
            x == z;/* no effect */
}
```

statement not reached, (W)

This message is produced when a statement will never be executed. This warning is usually associated with a conditional statement that is always false. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
#define FLAG 0
int x;

unreached_statement()
{

     if (FLAG &&  x)    /* FLAG is compile time constant 0 */
          x = 3;        /* never reached */
}
```

statement table is too large for available memory

The size of the statement table has exceeded the available memory.

storage class *storage_class* is invalid in file-scope, (E)

The **auto** or **register** keywords were used in a file-scope (external) declaration. The storage classes **auto** and **register** can only be used in a local scope (i.e., a function scope). These specifiers are not permissible on file scope variables.

Examples:

```
auto i;

register x;
```

storage classes *storage_class1* and *storage_class2* conflict, (E)

Two or more storage classes with conflicting meanings were given; the compiler does not discard any of the options and gives an error.

Example:

```
int foo() { static auto x; }
```

string too long (compiler limitation), (E)

The compiler cannot accept a string because it is too long.

struct or union type expected, (E)

Something other than a **struct** or **union** type was used with the "**.**" operator, or something other than a pointer to a **struct** or **union** was used with the "->" operator.

Example:

```
int foo() { int x, y; x=(&y).a; }
```

switch expression evaluated to constant *number*, (W)

This warning message is produced when an expression used in a **switch** statement evaluates to a constant. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
relation_op_evaluated_to_const4()
{
int i = FLAG;

      switch(i+2)
      {
      case 1: test1(); break;
      case 2: test2(); break;
      case 0: test0(); break;
      }
}
```

switch statement has no cases, (W)

The **switch** statement does not contain a case.

Example:

```
foo (i,j) {
      switch (0)              /* bad switch */
            i=9;
```

```
                        switch(i)               /* bad switch */
                        {
                              lab: break;
                        }
                        switch (j)              /* good switch */
                        {
                              case 1: break;
                        }
                  }
```


switch statements nested more than 16 deep (compiler limitation), (E)

The **switch** statements have been nested deeper than the compiler can accept.

Example:

```
int foo() {
      switch(0) switch(0) switch(0) switch(0) switch(0)
      switch(0) switch(0) switch(0) switch(0) switch(0)
      switch(0) switch(0) switch(0) switch(0) switch(0)
      switch(0) switch(0) switch(0) switch(0) switch(0);
}
```

symbol table is too large for intermediate file, (F)

This message indicates that the size of the symbol table exceeded available memory.

syntax error; unexpected *cause*, (E)

The compiler generates an error if correct C syntax is not followed in the file.

Example:

```
int c^D
```

Control-D indicates **End-Of-File**, which is unexpected at this point in the program.

[ANSI] tag "*tag_name*" is local to the prototype; match impossible, (W)

If you specify a structure tag name in the scope of a function prototype, the compiler will be unable to match the tag name in any call of the function. This message is issued if Microtec extensions are disabled on the command line (see the **-x** option in section *Enable Microtec Extensions* in Chapter 2, *UNIX/DOS User's Guide*, for further information about disabling Microtec extensions).

Example:

```
#pragma option -nx
```

```
#pragma option -nx
void g(void) {
      void f (struct NAME { int i; } *a); /* warning */
}
```

[ANSI] tag "*tag_name*" is local to the prototype; promoted to file scope, (W)

If you specify a structure tag name in the scope of a function prototype, the tag is promoted to the file scope. This message is issued if Microtec extensions are enabled. Refer to the **-x** option in section *Enable Microtec Extensions* in Chapter 2, *UNIX/DOS User's Guide*, for further information on enabling Microtec extensions.

Example:

```
#pragma option -x
void f(struct tag *s); /* warning */
```

In this example, `tag` is specified in the function prototype of function `f`. The function `tag` is promoted to the file scope.

[ANSI] too few arguments, (E)

A function was called with too few arguments.

Example:

```
int foo(int x, int y) { int a; foo(a); }
```

[ANSI] too many arguments, (E)

A function was called with too many arguments.

Example:

```
int foo(int x) { int a,b; foo(a,b); }
```

too many files opened, (F)

The preprocessor can open at most 63 files. This message indicates that too many files were opened at the same time by the preprocessor.

too many initializers for *variable*, (E)

According to the declaration, too many initial values have been specified for a variable.

Example:

```
int foo() { int a[4] = {1,2,3,4,5,6}; }
```


number trailing padding byte added (size=*size*), (W)

One byte of padding has been added to align a structure. The final byte size of the structure is indicated by *size*.

Example:

```
packed struct {
      char c;
      long x;
};
```

To evenly align the structure, the compiler will add one byte after `c` to create a structure with a final size of 6 bytes.

number trailing padding byte[s] added (size=*size*); use *option*, (W)

One or more (*number*) padding bytes has been added to align a structure. The final byte size of the structure is indicated by *size*. For information on compiler options that affect structure alignment, refer to the section *Modify Alignment* in Chapter 2, *UNIX/DOS User's Guide*.

Example:

```
packed struct {
      char c;
      long x;
};
```

To evenly align the structure, the compiler will add one byte after `c` to create a structure with a final size of 6 bytes.

type used as argument; replaced with its sizeof: *size*, (W)

An object type is incorrectly being used as an argument; it is replaced by its size.

Example:

```
int foo() { bar(int); }
```


unclosed character literal, (W)

A literal character should be enclosed by two single quotes (*'char'*). This message is emitted if one of the quotes of the literal character is missing. This message is from the preprocessor.

Example:

```
int a = ';
```

unclosed comment, (W)

The end of the source file was reached without finding a closed comment symbol (*/). This message is from the preprocessor.

Example:

```
int some_data;
/*
     This is an unclosed comment stream
<EOF>
```

unclosed string literal; treated as empty string, (W)

A string literal is missing a double quote (") to close the literal string. This message is from the preprocessor.

Example:

```
void main() {
     char *str = "abc;
}
```

undeclared identifier *identifier*, (E)

An identifier was used without a prior declaration.

Example:

Example:

```
int foo() { x=5;}
```


undefined enum *identifier*, (W)

If an identifier has been declared to be a type of an undefined enumerator and it is subsequently used in an expression, the compiler generates this warning.

Example:

```
enum xx yy; int foo() { yy = 0; }
```

undefined label *label*, (E)

A label was used but not defined within a function.

Example:

```
int foo() { goto x; }
```

undefined statement label "*label_name*," (E)

This message is issued if an undefined label was supplied in the **goto** statement.

Example:

```
void f(void)
{
      goto L1;
}
```

unexpected *symbol*; *symbol* inserted, (W)

An unexpected symbol was encountered, and the expected symbol was inserted to correct the situation. For example, if an **End-Of-Line** symbol is encountered before the ")" symbol while processing the "( )" expression, a ")" symbol will be inserted. This message is from the preprocessor.

Example:

```
"if (1
```

```
#if (1
```


[ANSI] unexpected symbol *symbol*; rest of line ignored, (I)

An unexpected symbol was encountered in processing the **#if/#ifdef/#include** etc. directives. The unexpected symbol causes the rest of the line containing the directive to be ignored. This message is from the preprocessor.

Example:

```
#if 1)
```

unintuitive sizeof value: *size*, (I)

If you cast the type of a **sizeof** operand to a different type, this message is issued. *size* indicates the size of the operand.

Example:

```
char c;
sizeof ((int) c);
```

unknown directive *directive*; ignored, (W)

The specified directive is not recognized. This message is from the preprocessor.

Example:

```
#stop
```

unknown option *option*, (W)

The compiler does not recognize the option entered on the command line.

unknown value "*unknown_suboption*" given for *option*, (W)

The compiler does not recognize the suboption entered on the command line.

Example:

```
-Fz
```


unpacked union "*union*" inside a packed struct (W)

This warning is generated when there is an unpacked union or struct nested inside a packed structure.

Example:

```
packed struct a {
      int i;
      struct b {          /* Unpacked struct in a packed */
            int i;        /* struct */
            int j;
      }

}
```

The compiler generates the nested structure as an unpacked structure.

Unresolved forward reference to struct/union type index *nnn*, (W)

When the debug option (**-g**) is specified and the program has complicated forward referenced **struct** types, this warning is issued. The situation is nonfatal to the linker and the XRAY Debugger.

use of "long double" conflicts with "no ansi" option; "double" assumed, (W)

The **long** and **double** specifiers conflict if the file has been compiled with the ANSI features disabled. The **long double** specifier is supported under the ANSI option (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA
long double i;
```

use of prototypes conflicts with "no ansi" option, (W)

use of prototypes conflicts with "no ansi" option, (W)

Function prototypes are ANSI extensions. Your file has been compiled with the ANSI features disabled, but the compiler found a function with a prototype specification. Function prototypes are supported under the ANSI option (see the **-A** option in section *Set ANSI-Compliant Mode* in Chapter 2, *UNIX/DOS User's Guide*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA
int foo(void);
```

value returned from a void function, (E)

A value was illegally returned from a **void** function.

Example:

```
void foo() { int x; x=5; return(x); }
```

variable *variable* has unknown size, (E)

A variable with an incomplete type was declared.

Example:

```
int foo() { struct tag x; }
```

variable *variable* may be used before set, (W)

This warning message indicates that a variable has been used before it has been defined. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking* in Chapter 2, *UNIX/DOS User's Guide*, for further information).

Example:

```
var_used_before_set()
{
int i,j,k;

        while(j)        /* j is not initialized in the */
                        /* first iteration of the loop */
        {
                if (i)  /* i is used before set */
                {
                        j = test();
                }
                else
                {
                        k = test();
                }
                test2();
        }
        test(i,j,k);    /* i is used before set */
}
```

variable *variable* replaced by constant *number*, (I)

This message is produced when a local variable is used but can be replaced by a constant for some assignments.

Example:

```
const_prop()
{
int i;

      i = 3;
      x = i + 2;        /* i == 3 */
      i = test();
      test2(i);
}
```

In this example, the variable `i` will be replaced by the constant `3`.

variable *variable* set but not used; assignment eliminated, (I)

This informational message is issued if a variable has been defined but is never used. The compiler will remove this variable.

Example:

```
unused_definition_elimination()
{
int i,j;

      i = xxx();
      j = yyy();  /* The value of the assignment of j is */
                  /* not used */
      if (i)
      {
            j = zzz();
            ddd(j);
      }
}
```

[ANSI] void pointer treated as char pointer, (W)

A pointer to **void** has been illegally used as an operand.

Example:

```
int foo() { void *p; p++; }
```

void type

This message is issued if you supply a bad identifier and declare the type of the identifier as **void**.

Example:

```
void ();
```

void type returned, (E)

A value of **void** type was returned from a function.